Optimization of PRAM-Programs with Input-Dependent Memory Access*

Welf Löwe

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, 76128 Karlsruhe, Germany.

Abstract. There exist transformations of PRAM programs with predictable communication behavior to existing architectures. We extend the class of tractable programs to those with communication depending on the input. First, we define this class of programs. Second, we give source code transformations to simplify the programs and to eliminate indirect addresses and conditionals. Third, we show how to derive the communication behavior statically. Fourth, we show how to compute the mapping at compile time. Finally, we give upper time bounds for execution on existing architectures.

1 Introduction

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 todays parallel programs highly depend on the architectures they are planed to run on. Designing efficient parallel programs is a difficult task that can be performed by specialists only. Porting those program to other parallel architectures is nearly unpossible without a considerable losts 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.

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. Several types of PRAMs are distinguished by their ability to access in parallel the same memory location. For an overview, see [KR90]. In this paper we exclude concurrent writes. Most parallel algorithms are designed for flavors of the PRAM models. 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.

* An extended version of the paper can be obtained via "World Wide Web": http://i44www.info.uni-karlsruhe.de/~loewe

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 real parallel machines are practically expensive, although theoretically optimal results exist [Val90]. The reason is the expensive synchronization, communication latency, communication overhead, and network bandwidth. In the LogP machine [CKP+93], these communication costs are taken into account. However, the number of processors are constant w.r.t. the problem size, and the synchronization must be programmed explicitly. The architecture dependent parameters of the LogP machine are the following. The *communication latency* L is the time a (small) message requires from its source to its destination. Observe that L is an upper bound on all source-destination pairs. The *communication overhead* o 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. The *gap* g is the reciprocal of the communication bandwidth per processor. It means that when a processor has sent (or received) a message, the next message can only be sent or received after time g. The *number of processors* P is the last parameter. These parameters have been determined for the CM-5 in [CKP+93] and for the IBM SP1 machine in [DMI94]. Both works found the prediction on expected running times of programs on these machines confirmed by practice.

However, designing programs directly for distributed, asynchronous machines is a difficult task. Usually, it can be performed by specialists only. The programs are often very complicated, not understandable, and not portable without a dramatic lost of performance. It is therefore beneficial to develop a method transforming programs for the PRAM into distributed programs in a systematic way to ensure correctness. For this task, two main steps are necessary, to transform the synchronous program into an equivalent asynchronous program, and to distribute the shared memory to particular processes.

We define the class of *non-oblivious* programs. These programs have no predictable communication behavior. Many massively parallel algorithms[2] belong to this class, e.g. almost all parallel algorithms on graphs are non-oblivious. We show that non-oblivious algorithms can be transformed into a LogP program such that the execution time is optimal within the LogP model. However, the transformation itself may require much time. Therefore, we give some efficient, but non-optimal transformations. Furthermore, we prove bounds for the expected running time of the resulting programs.

2 Classification of Parallel Programs

For classifying parallel programs 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

[2] A program is *massively parallel* iff the number of required processors increases with the size of the input.

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. We say that processor i *communicates at time* t *with processor* j iff there is 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 treated as assignments but without writing into the shared memory.

Definition 1. 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

$$V_{A,x} = \{\langle i, t\rangle : 0 \leq i < P_A(n) - 1, 0 \leq t \leq T_A(n)\},$$

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

A *communication scheme* of a PRAM algorithm A for inputs of size n is a directed acyclic graph $G^*_{A,n}$, that is the union of all communication structures of A with a valid input of size n.

Definition 2. A parallel algorithm is called *oblivious* iff its communication structure and its communication scheme are the same for all inputs the same size. Otherwise, it is called *non-oblivious*.

Important problems where non-oblivious implementations may be desired are e.g. operations on sparse matrices, adaptive multi-grid-methods for the numerical solution of partial differential equations, graph algorithms. Observe that none of these algorithms is implemented to work in parallel. In fact, engineers prefer to implement these problems mainly sequentially as the sequential execution time is currently better than the execution time of a parallel implementation. Even Valiants PRAM simulation [Val90] that yields theoretically optimal results requires approximately $10 - 20ms$ for one PRAM step on a MASPAR [ZK93].

3 Deriving Communication Schemes

In this section we show how the communication scheme can be derived statically. From definition 1 it follows immediately that the communication structure of an oblivious program only depends on the size of the input and its communication scheme is equal to this communication structure. If we knew the size of the input and the fact that the program is oblivious, we could derive the communication structure just by a (synchronous) sample execution of the program.

If the program is non-oblivious there are some communications depending on the input of the program. In this case we assume that all possible communication occurs. All further transformations are based on the following assumptions: First, the input size n is known at compile time[3], and second, the program representation is a control flow graph. We assume that the control flow graph is a directed graph $CFG = (V, E)$ whose vertices are program points and $(v_1, v_2) \in E$ iff there is a direct control flow from v_1 to v_2. There are two types of vertices: AND-vertices arise from a **pardo**, i.e. the control flows to all successors, and OR-vertices arising from loops or conditional statements, i.e. the control flows to one successor.

First, we show how to decide whether a parallel program is oblivious or not. While deciding this we simplify the program. Second, we include a transformation making some non-oblivious programs oblivious. Finally, we derive the communication scheme from the simplified, transformed program. The basic techniques applied on oblivious programs can be found in [ZL94].

3.1 Simplifying PRAM Programs

Our goal is to transform the *CFG* into an acyclic directed graph whose vertices are AND-vertices by loop unrolling, procedure inlining, and recursion elimination[4], see algorithm 1. The size of the resulting code is limited by the work done by the PRAM algorithm. Therefore, the size of the code is at most $O(T(n) \times P(n))$. Hence, if the original algorithm is polynomial in the required time and processors, the size of the resulting code is also polynomial.

Lemma 3. *Let P be a parallel program. If constant folding, loop unwinding, recursion elimination, and procedure inlining is successfully applied to P, then the control flow graph of the transformed program is acyclic.*

Proof. As there is no loop and no recursion in the transformed program, the corresponding *CFG* must be acyclic.

[3] For many applications this is no restriction (assume e.g. the data is generated by sensors). For others there exist design methods for reducing the required number of processors to a constant (as e.g. for all divide & conquer algorithms). As a side effect, this constant equals to the input size of the resulting program. Finally, if the input size is bound by an upper and a lower limit it is possible to perform the following optimizations for all sizes in between these limits. If $G^*_{A,n} \subset G^*_{A,m}, m > n$ the optimizations can be implemented efficiently, i.e. schemes needn't be optimized redundantly.

[4] For non-recursive procedures and functions, procedure inlining is no problem. When all recursions can be eliminated, then procedure inlining is no problem. Sometimes it is not possible to check statically the number of recursions or iterations, even if it is constant or a function of the input size that is constant, as well. However, we believe that this case doesn't happen too often in practice. In Fortran-programs most of the loops are for-loops. For non-for loops and recursion techniques like [FSZ91, Zim91] of automatic complexity analysis can be used to derive the number of iterations of a loop and recursive calls of a recursive procedure, respectively.

Algorithm 1. *Simplify a PRAM Program and Check Obliviousness.*

(1) apply constant folding to P;
(2) eliminate recursion from P;
(3) unwind loops in P;
(4) inline all procedures of P;
(5) **repeat**
(6) apply constant folding to P;
(7) eliminate dead code from P;
(8) **until** P does not change;
(9) compute control flow graph (*CFG*) for P;
(10) mark each vertex v if v is not **pardo**;
(11) compute for each marked v:
(12) number m_v of marked vertices;
(13) on the longest path to v in *CFG*;
(14) label each marked v with t_v where:
(15) PRAM completion time $t_v = m_v + 1$;
(16) **if** *CFG* has an OR-vertex $\vee$ *CFG* contains indirect addressing
(17) **then** output *CFG* and P ***is not oblivious***;
(18) **else** output *CFG* and P ***is oblivious***;
(19) **fi**;

3.2 Reducing the Non-obliviousness

After the above transformations a parallel program may contain indirect addresses on the left side of assignments, as e.g. in Tree Contraction. Suppose processor i executes an assignment $a[f(a,i)] := expr$ where *expr* is an expression and $f(a,i)$ is an index function without side-effects (especially it doesn't raise exceptions like a division by 0). We transform the above assignment into:

```
distributed

forall j := 0 to |a| − 1 do in parallel
    if j = f(a, i) then
        a[j] := expr;
    end; −if
end; −forall
```

where j doesn't occur neither in $f(a,i)$ nor in *expr* and $|a|$ is the size of the shared address space. We add this transformation after line (4) of algorithm 1[5].

Lemma 4. *Let P be a simplified PRAM program. The above transformation of P doesn't change its semantics.*

[5] This transformation doesn't make the program more oblivious since the conditional statement cannot be eliminated by constant folding because $f(a,i)$ is not a constant. With a similar transformation we could eliminate the indirect addresses on the right side of assignments, as well. We don't do so. As we will see, these indirect addresses do not cause an all to all communication in the resulting distributed asynchronous program. Hence, it doesn't make sense to waste processors or time.

Proof. Because $f(a,i)$ is a function without side-effects we can call it arbitrarily often. This function $f(a,i)$ returns an index of a. There are only $|a|$ many possible return values of $f(a,i)$. Hence, we can check all indices if they are equal to the return value of $f(a,i)$ and execute the assignment for the index equal to this value. There are no dependencies between all these checks, hence, we can execute them in parallel.

Note, that a program running on p processors before this transformation requires at most $p \times |a|$ processor afterwards. Of course, instead of the **pardo** statement we could use a **for** loop. In this case our algorithm doesn't require more processors but time $t + |a|$ if t is the running time of the original program. However, we save an all to all communication in the resulting asynchronous program.

A parallel program may contain a conditional checking some predicate on the state of the shared memory. E.g. the Game of Life checks how many of the neighbor cells are dead. These conditionals can be removed by pessimistic assumption that each branch has to be computed for computing the conditional statement [ZL94].

3.3 Deriving Communication Schemes

After the transformations given in the last sections the control flow graph of PRAM programs only contain **pardo** and assignment vertices. The only remaining source of non-obliviousness in these programs is indirect addressing on the left hand side of assignments.

Algorithm 2. *Compute a Communication Scheme.*

$\mathbf{t = 0}:$ $G_0 = (V_0, E_0)$ where $V_0 = \{(a[i], 0) : 0 \leq i < p\}$ and $E_0 = \emptyset$

$\mathbf{t > 0}:$ $G_t = (V_t, E_t)$ where $V_t = V_{t-1} \cup V_t'$ and $E_t = E_{t-1} \cup E_t' \cup E_t''$

$$V_t' = \{(x,t) : \exists v \in CFG : t_v = t \ \wedge v \text{ contains } x := c\}$$

$$E_t' = \{((y,t'),(x,t)) : \exists v \in CFG : t_v = t \ \wedge v \text{ contains } x := c \wedge y \text{ is operand in } c \ \wedge t' = \max\{\bar{t} : (y,\bar{t}) \in V_{t-1}\}$$

$$E_t'' = \{((a[i],t'),(x,t)) : 0 \leq i < p \ \wedge \ existsv \in CFG : t_v = t \ \wedge v \text{ contains } x := c \ \wedge a[f(a)] \text{ is operand in } c \wedge t' = \max\{\bar{t} : (a[i],\bar{t}) \in V_{t-1}\}\}$$

Let $G^* = G_m$ if m is the highest label of all vertices in CFG.

Theorem 5. *Let P be a simplified PRAM program without indirect addressing on the left hand side of assignments and without conditionals. Algorithm 2 computes a graph containing all vertices and edges of the communication scheme G^* of P.*

Proof. The proof is by induction on the steps of the PRAM program: G_0 is computed correctly since it contains a vertex for each memory cell. There is no communication at PRAM step 0. Therefore, E_0^* is empty. We assume that G_{t-1} is computed correctly. Vertices in the CFG labeled with t correspond to the PRAM assignment executed at time t. This labeling is well-defined since the CFG is acyclic by lemma 3. Hence, for each PRAM processor that executes an

assignment at time t a vertex v is added to G_t. If a memory cell is read in the left hand side of the assignment an edge from the vertex u corresponding to the last assignment that wrote this cell to v is added to the set of edges of G_t. Hence, all oblivious communication to vertices v corresponding assignments executed at time t have a corresponding edge in G_t. If the left side of an assignment executed at time t contains an indirect address, edges from all vertices corresponding to the last write accesses to each memory cell is added. Hence, for all non-oblivious communications that possibly occur at time t there exist an edge in G_t. Therefore, G_t is a communication scheme of the PRAM program executing the first t steps of P. Let m be the highest label in CFG. There is no assignment executed at time $t > m$. Therefore, the construction of G terminates after m steps. That completes the proof.

4 Compiling Non-oblivious Programs

For transforming PRAM programs to LogP programs two tasks must be performed: first consideration of the asynchronous execution model and second, the distribution of the shared memory. We construct the communication scheme where every node corresponds to an assignment:

$$a[f(i,a)] := \Phi(a[f_1(i,a)], \ldots, a[f_m(i,a)]).$$

4.1 Program Transformation

First, we discuss the index functions in more detail. Therefore, we number the nested indirections in addressing. Observe, that all nestings of indirect addresses end up in a direct address, i.e. with a index function independing of the memory's state. We extend the above assignment by the number of nested indirect addresses:

$$a[f_0^0(i,a)] := \Phi(a[f_1^0(i,a)], \ldots, a[f_m^0(i,a)])$$

where $f_k^j(i,a)$ is defined as:

$$\Psi_k^j(i, a[f_{k,1}^{j+1}(i,a)], \ldots, a[f_{k,n}^{j+1}(i,a)])$$

if index expression Ψ_k^j contains further indirect addresses. Otherwise it simplifies to $f_k^j(i)$. Since for all nested indirect addresses there exist direct addresses, the communication structure can be computed partially. For oblivious communications of a non-oblivious program sender and receiver can be computed at compile time. For non-oblivious communications they have to be computed at run time. After having received all data the expression Φ can be computed. If the left hand side of the assignment doesn't contain indirect addressing the result can be sent via a channel to the next process as in oblivious programs. If it does, the next process has to be computed at run time and a message has to be sent to this

process. We assume that there is no indirect addressing on the left side of the assignments. Hence, an assignment A has the form:

$$a[f_0^0(i)] := \Phi(a[f_1^0(i,a)], \ldots, a[f_m^0(i,a)]).$$

The necessary transformations of an assignment A executed at PRAM-time t is shown below.

Algorithm 3. *Remove Indirect Addresses.*

(1) **repeat**
(2) substitute $a[f_k^j(i,a)]$ with $process[f_k^j(i,a)].get\_arg(f_0^0(i),t)$;
(3) substitute $f_k^j(i,a)$ with $\Psi_k^j(i, a[f_{k,1}^{j+1}(i,a)], \ldots, a[f_{k,n}^{j+1}(i,a)])$;
(4) **until** A does not change;
(5) **repeat**
(6) substitute $a[f_k^j(i)]$ with $a_{f_k^j(i)}$;
(7) **until** A does not change;

Process $a[f_0^0(i)] := \Phi(a[f_1^0(i,a)], \ldots, a[f_m^0(i,a)])$.

(1) **process** $(f_0^0(i),t)$ – *simulates A executed by processor $f_0^0(i)$ at time t*
(2) **entry** *start* **is**
(3) $[a_{f_1(i)} \cdots a_{f_m(i)}] :=$[**recv**$(f_1(i),t_1) \ldots$ **recv**$(f_m(i),t_m)$]
– – $comm(f_0^0(i), t, f_{1\cdots m}(i), t_{1\cdots m})$
(4) A transformed by algorithm 3;
(5) [**send**$(a_{f_0^0(i)}, (\iota_1,\tau_1)) \cdots$ **send**$(a_{f_0^0(i)}, (\iota_n,\tau_n))$]; – $-comm(\iota,\tau, f_0^0(i), t)$
(6) **end** *start*;
(7)
(8) **entry** $get\_arg(i,\tau)$ **is**
(9) **send**$(a_{f_0^0(i)}, (i,\tau))$;
(10) **end** *get_arg*;
(11) **end** – – **process** $(f_0^0(i),t)$

Each of the transformed assignments A is associated to a single process $(f_0^0(i),t)$. Additionally, the initial state of each memory cell must be provided by separate processes. For oblivious communications we use a channel-oriented model. We assume that each edge in the communication structure is a channel through which communication has to go. The semantics of $recv(f_x(i),t_x)$ is that the process receives the data sent by process $(f_x(i),t_x)$. The receiving of data and the assignment to the local variables $a_{f_1(i)} \cdots a_{f_m(i)}$ is executed in arbitrary order. The $send(data,(\iota,\tau))$ operation is dual to the receive operation, i.e. process (i,t) sends *data* to process (ι,τ). A *recv* operation is terminated if it received the data. Furthermore, we assume that processes can send without any synchronization with the receiving processes, and that the channels are safe.

Lemma 6. *Let P be a simplified PRAM program. After having applied algorithm 3 to P it doesn't contain any indirect access to the shared memory.*

Proof. The array *process* will contain the addresses of all processes that possibly contain an argument for computing A. In contrast to array a, array *process* can

be computed statically, since we know all processes at compile time. The length of *process* is at most equal to the length of a. The message $get\_arg(f_0^0(i), t)$ sent to a process returns the argument stored in this process to the sender, i.e. to process $(f_0^0(i), t)$ (line 4). The index functions f can be computed at compile time since they depend only on i and not on the array a (line 12).

Theorem 7. *Let P be a simplified PRAM program. The processes $(f_0^0(i), t)$ can be executed asynchronously and with memory. They compute the same function as the original PRAM program did.*

Proof. Assuming that constant folding, loop unrolling and recursion elimination have been applied complete and correctly to P. Then P has a finite number of statements, because of lemma 3. Therefore, all indirect addresses on the left hand side of assignments and all conditionals can be eliminated. Because of lemma 4 these transformations are correct also. Therefore, the transformed PRAM program is semantical equivalent to the original one. By lemma 6 all accesses to the shared memory can be eliminated. Hence, each process must only contain the array cell that it writes. Therefore, the shared memory can be distributed. The correctness of the asynchronous execution is proven by induction on a topological ordering of the vertices in the communication scheme G^* of P. By theorem 5 G^* can be computed statically and contains all possible data dependencies of P. A vertex $(i, 0)$ in G^* corresponds to a process providing $a[i]$ at time 0. A vertex $(i, t), t > 0$ in G^* corresponds to a process $(f_0^0(i), t)$. We label each vertex that has been computed as finished. We assume that processes $(i, 0)$ contain the correct initial value of the array cell $a[i]$, i.e. all vertices with depth 0 in G^* can be labeled initially. A vertex v in G^* can be computed if all data needed in v is available. By induction hypothesis all predecessors of v are labeled (oblivious communication) and have sent the correct values to v (oblivious communication). Hence, the processors containing further values needed in v can be computed. Also by induction hypothesis all potential predecessors of v are labeled (non-oblivious communication). Therefore, v can order these values and computes correctly. After computation it sends the correct value to its (oblivious) successors and provides this value for further (non-oblivious) requests. v can be labeled as finished, as well.

Lemma 8. *Let d be the maximal depth of indirect addresses in a program P and dg the maximal degree of vertices in $G^*(P)$. The over all time L_{max} for communication from a vertex to its direct successor (in G^*) is at most:*

$$L_{max} \leq ((1 + 2d) \times (L + 2o + (dg - 2) \times max[o, g]).$$

Proof. Assume that a message is the last to be sent from a vertex. Then there are $out - degree - 1$ messages sent before. Hence, $out - degree - 1$ gaps have to be guaranteed. Then, sending the message takes time o and the communication delay is L. The same holds for receiving the message where in-degree -1 gaps have to be guaranteed. After one necessary oblivious communication for each nested indirect addressing two messages have to be sent in sequence.

Theorem 9. *Let P be a parallel program whose communication scheme G^* has diameter T. Let C be the maximal computation time for the vertices in G^*. Its execution time is at most: $T_{max} \leq (T-1) \times L_{max} + T \times C$.*

Proof. On the longest path T tasks have to be computed, $T-1$ communications occur sequentially. Each computation requires at most time C, each communication requires at most time L_{max}. There is no waiting necessary because we chose the longest path.

4.2 Program Optimization

Merging some of the processes into one processor saves time required for communication. On the other hand, the degree of parallelism is decreased. In fact we *cluster* the computations done in the vertices of a communication scheme onto the processors of the LogP machine such that the execution time is minimal. We can implement any oblivious PRAM program as an optimal LogP program w.r.t. its computation time. However, the transformations themselves are exponential. In [PY90] Papadimitriou and Yannakakis showed that finding an optimal clustering is NP-hard, even if $o = g = 0$ and $P = \infty$. We can therefore not expect to find an efficient and optimal transformation. It is also known that approximative solutions which are better than $2 \times T_{optimal}(G)$ cannot be found in polynomial time when $o = g = 0$, unless $P = NP$. However, in [LZ95] it is proven that the optimal solution can be found in polynomial time if G 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 $o = g = 0$. All these works assumed oblivious algorithms. However, we can extend the results to non-oblivious algorithms. Therefore, we define the notion of granularity of communication schemes. Informly speaking, the granularity of a communication scheme G^* is the ratio of computation and communication costs in G^* on a distinct parallel machine.

Definition 10. Let $PRED_v$ be the set of all direct predecessors u of a vertex v in a communication scheme G^*. Let $L_{max}(u,v)$ be the maximal overall communication cost for sending a message from vertex u to vertex v (including overheads and gaps). Let C_u be the time for computing u. The *granularity of a vertex* is defined as

$$g(v) = \frac{\min\limits_{u \in PRED_v} \{C_u\}}{\max\limits_{u \in PRED_v} \{L_{max}(u,v)\}}.$$

and the granularity of a communication scheme is defined as $g(G^*) = \min\limits_{v \in G^*} \{g(v)\}$. G^* is *coarse grained* if $g(G^*) \geq 1$, otherwise it is called *fine grained.*

Note, that the granularity of a communication schemes can be computed at compile time even if the program is non-oblivious. In contrast to other more

qualitative definitions of granularity we can give upper bounds for the execution time of a program on a parallel machine in terms of the granularity of the corresponding communication schemes.

Theorem 11. *Any clustering of a communication scheme G^* computing only vertices of the same path in G^* on one processor leads to a program running in at most:* $T_{cluster} \leq (1 + \frac{1}{g(G^*)}) \times T_{optimal}(G^*)$.

Proof. Let $\mathcal{P}$ be the path from a vertex v_i with $idg_{v_i} = 0$ to a vertex v_o with $odg_{v_o} = 0$ with the maximal sum of computation times of its vertices and communication delay between these vertices. For the optimal clustering of G^* it holds: $T_{optimal}(G^*) \geq \sum_{v \in \mathcal{P}} C_v$. Let $L_{max}(v) = L_{max}(u, v)$ where u is the immediate predecessor of v in path $\mathcal{P}$. Set $L_{max}(v_i) = 0$. For the naive implementation of G^* it holds:

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

Because a clustering along the paths in G^* cannot increase the running time compared to the naive implementation, the proof is complete.

Note, that for coarse grained communication schemes G^* the time bound reduces to $2 \times T_{optimal}(G^*)$ even if $o \neq 0, g \neq 0$, and the program is non-oblivious. Heuristic solutions for clustering communication structures where $g \neq 0$ and $o \neq 0$ can be found in [ZL94]. The techniques for clustering communication structures can be applied to communication schemes without any modifications.

5 Conclusions

We showed for another 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 LogP machine. This class of *non-oblivious* parallel program is characterized by their communication behavior which varies for inputs of size n. From a practical point of view this class is large. It contains for example basic techniques as for instance pointer jumping, pebble game and tree contraction. Almost all parallel algorithms on graphs are non-oblivious. If the the input size of a program is limited before compiling, its communication scheme can be derived statically. With this information, we are able to include the transformation from non-oblivious PRAM programs to distributed programs running existing parallel machines into compilers. Therefore, the clustering and scheduling algorithms for mapping oblivious PRAM

programs onto the LogP machine can be applied for non-oblivious PRAM programs, as well. Hence, for correct execution of non-oblivious PRAM programs, it is not necessary to perform expensive the PRAM-simulation. Additionally, we can give upper time bounds for execution for the resulting programs on existing architectures. Future work will consider the behavior of clustering and scheduling heuristics applied to the communication schemes of several graph algorithms. Beside the upper time bounds it would be interesting to determine average running times of the resulting programs. Therefore, we want to label the non-oblivious communications with the possibility of their occurrence.

References

[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.

[DMI94] 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.

[FSZ91] P. Flajolet, B. Salvy, and P. Zimmermann. Average case analysis of algorithms. *Theoretical Computer Science*, 1991.

[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 *Handbook of Theoretical Computer Science Vol. A*, pages 871–941. MIT-Press, 1990.

[LZ95] W. Löwe and W. Zimmermann. On finding optimal clusterings of task graphs. In *Aizu International Symposium on Parallel Algorithm/Architecture Synthesis*. IEEE Computer Society Press, 1995.

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

[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.

[Zim91] Wolf Zimmermann. The automatic worst case analysis of parallel programs: Simple parallel sorting and algorithms on graphs. Technical Report TR-91-045, International Computer Science Institute, August 1991.

[ZK93] Wolf Zimmermann and Holger 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.