Scheduling Iterative Programs onto LogP-Machine

Welf Löwe and Wolf Zimmermann

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe,
76128 Karlsruhe, Germany,
{loewe|zimmer}@ipd.info.uni-karlsruhe.de

Abstract. Usually, scheduling algorithms are designed for task-graphs. Task-graphs model oblivious algorithms, but not iterative algorithms where the number of iterations is unknown (e.g. while-loops). We generalize scheduling techniques known for oblivious algorithms to iterative algorithms. We prove bounds for the execution time of such algorithms in terms of the optimum.

1 Introduction

The communication behavior of many data-parallel programs depends only on the size of their inputs. If this size is known in advance, it may be exploited for translation and optimization to improve the efficiency of the generated code [13]. In contrast to general compiling techniques using data-dependency analysis [12, 10, 5, 11], all synchronization barriers can be removed and data as well as processes can be distributed automatically. Hence, programmers may focus on the inherent parallelism of the problems and ignore the parameters of target machines. More specifically, programmers can use a synchronous, shared memory programming model. Neither data alignment nor mapping of processes onto processors is required in the source code[1]. The LogP-machine [1] assumes a cost model reflecting latency of point-to-point-communication in the network, overhead of communication on processors themselves, and the network bandwidth. These communication costs are modeled with parameters *L*atency, *o*verhead, and *g*ap. The gap is the inverse bandwidth of the network per processor. In addition to L, o, and g, parameter P describes the number of processors. The parameters have been determined for several parallel computers [1, 2, 3, 4]. These works confirmed all LogP-based runtime predictions.

However, among those problems that are not oblivious, we identify another class of high practical interest: problems that iterate an oblivious program where the number of iterations is determined at run time. We call those problem *iteratively oblivious*. The Jacobi-Iterations for solving linear equation systems or solvers based on the method of conjugated gradients (CG-solvers), e.g., are iteratively oblivious. The present work shows how to compile those programs to

[1] Equivalent to the PRAM-machine model, see [6].

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 332–339, 1999.

LogP-architectures, how to optimize them on a specific instance of the LogP-machine, and proves time bounds for their execution in terms of the theoretical optimum.

The paper is organized as follows: Section 2 introduces the basic definitions. Section 3 defines translations of iterative oblivious programs to LogP architectures. Section 4 proves time bounds of the resulting programs.

2 Basic Definitions

2.1 Oblivious and Iterative Oblivious Programs

First, we define the relation between PRAM-programs $\mathcal{P}$ and task-graphs $G_x = (V_x, E_x, \tau_x)$ for inputs x. The basis is a PRAM-model with an arbitrary number of processors with local memory, a global memory and a global clock. The task-graph of a PRAM-program on input x is basically the data-flow graph w.r.t. input x. More precisely, we partition the program on each processor into phases with the following structure: (1) read data from global memory, (2) perform computation without access of global memory, and (3) write data to global memory. Each of these phases becomes a task whose weight is the execution time of its computation. The task-graph G_x has an edge (v, w) iff phase v writes a value into the global memory that is read by w, i.e. there is data-flow from v to w via the global memory. The execution time $T_{\mathcal{P}}(x)$ of PRAM-program $\mathcal{P}$ on input x is simply the weight of the longest weighted path of G_x. The work $W_{\mathcal{P}}(x)$ is the sum of all task weights. A PRAM-program $\mathcal{P}$ is *oblivious* iff $G_x = G_y$ for any two valid inputs x and y of the same size. For simplicity we assume that we operate only on arrays of floating point numbers and the size is measured by the number of array elements. For oblivious programs, we write G_n for the task-graph for inputs of size n. A PRAM-program $\mathcal{P}$ is *iteratively oblivious* iff (i) it is oblivious, (ii) a sequential composition of iteratively oblivious programs, or (iii) a loop over iteratively oblivious programs. We assume that the termination condition of a loop belongs to the body of the loop and the result of its evaluation is written into global memory. The class of iteratively oblivious programs is a generalization of oblivious programs.

2.2 LogP-Schedules

For scheduling task-graphs to LogP-programs, we have to define when and where computations, send-operations and receive operations have to be executed. A *LogP*-schedule for a task-graph $G = (V, E, \tau)$ is a mapping $s : Op(V) \to 2^{\mathbb{N}\times\mathbb{N}}$ compatible to precedence constraints defined by G and the LogP-model. $(i, t) \in s(o)$ means that operation o is scheduled on processor P_i for time t. The *execution time* $T(s)$ is the completion time of the last task computed by s. An *input task* is a task containing an input value, an *output task* is a task computing an output value.

In this paper, we discuss three scheduling strategies: The *naive transformation* simply maps every task onto a single processor and assumes that there

Algorithm 1

Process *for task* v *with predecessors* $u_1, \ldots, u_n$, *successors* $w_1, \ldots, w_m$, *and computation* $\Phi(u_1, \ldots, u_n)$

```
(1)   process P_v
(2)     do n times: m := recv; a_{m.src} := m.val;
(3)     res := Φ(a_{u_1}, ..., a_{u_n});
(4)     send(res, v, w_1), ..., send(res, v, w_m)
(5)   end
```

are enough processors available. Every task v is implemented by a process P_v which first receives messages from its predecessors containing the operands, then performs its computation, and finally sends the results to its successors. The messages contain a value, their source, and their destination. Algorithm 1 sketches the transformation.

A *linear clustering* computes a vertex-disjoint path cover of G (not necessarily a minimal one) and maps each path onto a separate processor. It also assumes that there are enough processors available.

The last scheduling strategy uses a layering of the task graph G, i.e. a partition λ of the vertices of G such that $\Lambda_0 = \{v : idg_v = 0\}$ and $\Lambda_{i+1} = \{v : PRED_v \subseteq \Lambda_{\leq i} \wedge v \notin \Lambda_{\leq i}\}$ where $\Lambda_{\leq i} = \Lambda_0 \cup \cdots \cup \Lambda_i$, $PRED_v$ is the set of direct predecessors of vertex v, and idg_v is the *indegree* of v. *Brent-clustering* schedules each layer separately onto P processors. The upper bound $L^{\max}_{L,o,g}(u,v)$ of the total communication costs between two vertices u and v is $(L + 2o + (odg_u + idg_v - 2) \times \max[o,g])$, cf.[13], where odg_u is the outdegree of u. We define the notion of granularity of a task graph G which relates computation costs to communication costs, for a task v it is $\gamma_{L,o,g}(v) = \min_{u \in PRED_v} \tau_u / \max_{u \in PRED_v} L^{\max}_{L,o,g}(u,v)$ The granularity of G is defined as $\gamma_{L,o,g}(G) = \min_{v \in G} \gamma_{L,o,g}(v)$. Note that the maximum communication costs and, hence, the granularty of a task-graph depend on the communication parameters L, o and g of the target machines. We summarizes the upper bounds on the execution time for scheduling task graphs – naive transformation: $(1+1/\gamma_{L,o,g}(G))$ cf. [13], linear clustering: $(1+1/\gamma_{L,o,g}(G)$ cf.[9], Brent-clustering: $(1 + 1/\gamma_{L,o,g}(G)(W(G)/P + T(G))$ cf. [8].

3 Implementing Iterative Oblivious Programs

Oblivious programs can be easily scheduled to LogP-machines if the input size is known: compute the task graph $G(n)$ for inputs of size n and schedule it according to one of the scheduling strategies for LogP-machines. This is not possible for iterative oblivious programs since it is unknown how often the loop bodies in the program are executed. The basic idea is to perform a synchronization

after a loop iteration. The oblivious loop bodies are scheduled according to a scheduling strategy for oblivious programs. Furthermore, it might be necessary to communicate results from the end of one loop iteration to the next or to the tasks following the loop. The transformation assume that the input size n is known. We first discuss the naive transformation and then demonstrate the generalization to linear clusterings and Brent-clusterings.

3.1 Naive Transformation of Iterative Oblivious Programs

The naive transformations of iterative oblivious program $\mathcal{P}$ for inputs of size n to LogP-programs works in two phases: The first phase computes naive transformations for the oblivious parts recursively over the structure of iterative oblivious programs, the second phase connects the different parts. The tasks determined by Phase 1 has the following property:

Lemma 1. *Every loop in an iteratively oblivious program, has a unique task computing its termination condition.*

Proof. Since the termination condition of a loop computes a boolean value, it is computed by a single task. Naive transformations of oblivious programs are non-redundant. Thus the task computing the termination condition is uniquely determined.

We identify loops with the tasks computing their termination condition. Phase 2 transforms the processes computed by Phase 1 for implementing $\mathcal{P}$. The processes computing tasks not belonging to a loop require no further transformation, because the preceding and succeeding tasks are always known are executed once. The tasks belonging to a loop can be executed more than once. The source of messages received by input tasks of a loop body depends on whether the loop is entered or a new iteration is started. However, the sources of the messages determine the operand of the computation of an input task. This mapping is implemented by the table *opd* and can also be defined for all tasks. Output tasks of loop bodies require a special treatment. Their sending messages can be classified as follows: the destination of the message is definitely a successor in the loop body (*oblivious messages*), the destination of the message depends on whether the loop is terminated or not (*non-oblivious messages*).

Because loops can be nested, an output task of an inner loop can also be an output task of an outer loop. Let $LOOPS_v$ be loops where v is an output tasks and $OUTER_v$ be the outermost loop in $LOOPS_v$. Then, every non-oblivious message can be classified according to their potential destination w.r.t. a loop in $LOOPS_v$: $FALSE_{v,l}$ is the set of potential successors if loop l does not terminate. $TRUE_{v,l}$ is the set of potential successors of the enclosing loop, if loop l does terminate. $DEFER_{v,l}$ is the set of potential successors outside of the enclosing loop. Obviously, it holds $DEFER_{v,OUTER_v} = \emptyset$. Thus, an output vertex v receives messages on termination of the loops $LOOPS_v$ and sends its messages to either $TRUE_{v,l}$ or $FALSE_{v,l}$ (lines (6)–(13)). Algorithm 2 summarizes this discussion. If the task is also an output task on the whole program, then the

Algorithm 2

Process *for task v belonging to a loop with oblivious successors $w_1, \ldots, w_m$, and computation Φ with n operands*

```
(1)   Process P_v
(2)     loop
(3)       do n times: m := recv; arg_{opd_{m.src}} := m.val;
(4)       res := Φ(arg_1, ..., arg_n)
(5)       send(res, v, w_1), ..., send(res, v, w_m);
          -- lines (6)-(13) only if v is output vertex
(6)       term := recv_term_message; l := term.src;
(7)       while term.terminated ∧ l ≠ OUTER_v do
(8)         for w ∈ TRUE_{v,l} do send(res, w);
(9)         term := recv_term_message; l := term.src;
(10)      end;
(11)      if term.terminated then
(12)        for w ∈ TRUE_{v,l} do send(res, w);
(13)      else for w ∈ FALSE_{v,l} do send(res, w)
(14)    end
(15)  end
```

Algorithm 3

Process *for task v computing the termination condition:*

```
(1)   process P_v
(2)     loop
(3)       do n times: m := recv; arg_{opd_{m.src}} := m.val;
(4)       res := Φ(arg_1, ..., arg_n)
(5)       broadcast(res, OUT_v);
(6)     end;
(7)   end;
```

process is terminated. For simplicity we assume that the termination messages arrive in order of their broadcast. Algorithm 3 shows the processes computing a termination condition. It must be broadcasted to all output vertices of the loop.

Theorem 1. *The naive transformation on the LogP-Machine is correct.*

Proof. (-sketch:) Let $\mathcal{P}$ be an iteratively oblivious PRAM-Program and $[\![\mathcal{P}]\!]$ be the LogP-program obtained from $\mathcal{P}$ by the naive transformation. We prove the correctness of the naive transformation by induction on the structure of $\mathcal{P}$.
Case 1: $\mathcal{P}$ is oblivious. The correctness follows directly from the results of [13].

Case 2 $\mathcal{P} \equiv \mathcal{P}_1; \mathcal{P}_2$. Suppose the naive transformation is correct for $\mathcal{P}_1$ and $\mathcal{P}_2$. Then the output tasks of $[\![\mathcal{P}_1]\!]$ send their results to the input tasks of $[\![\mathcal{P}_2]\!]$. If these messages are sent to the correct destination, then the naive transformation is also correct for $\mathcal{P}$, because the mapping *opd* assigns the operands correctly.

If $\mathcal{P}_1$ is not a loop, then the messages sent by the output vertices of $[\![\mathcal{P}_1]\!]$ are oblivious. Thus the messages of the output vertices are sent to the correct destination. Let $\mathcal{P}_1$ be a loop and v be an output task v. Since the termination conditions of all loops in $LOOPS_v$ must yield true, every input task w that is a potential successor of v satisfies $w \in DEFER_{v,l}$ for all $l \in LOOPS_v \backslash OUTER_v$, or $w \in TRUE_{v,OUTER_v}$ where the sets are constructed w.r.t. $\mathcal{P}$. By construction of these sets, the messages are sent to the correct destination (or are output vertices of $\mathcal{P}$).
Case 3 $\mathcal{P}$ is a loop with body $\mathcal{B}$. This case requires a proof by induction on the number of iterations of $\mathcal{P}$. The induction step can be performed with similar arguments as in Case 2.

Linear schedules and Brent-schedules can be implemented similarly. The first phase determines the processes obtained by linear clustering or Brent-clustering. The receiving phases and sending phases for task v is the same as in Algorithm 2 and occur also immediately before and after the computation performed by task v. In particular, if v is an output task, then non-oblivious messages are sent. If process corresponding to a cluster is in a loop body, then it is iterated as in Algorithm 2. We denote this transformations by $[\![\cdot]\!]_L$ and $[\![\cdot]\!]_B$, respectively.

4 Execution Time

The estimation on the execution time is done by two steps. First, we estimate the execution time if the broadcast performed per loop iteration can be executed in zero time. Then, we add the factor lost by this broadcast. For both cases,

Let $\mathcal{P}$ be an iterative oblivious PRAM-program, x be an input of size n, and $\mathcal{P}'$ obtained by one of the transformations of Section 3. $TIME_{\mathcal{P}'}(x)$ denotes the execution time of a LogP-program $\mathcal{P}'$ on input x. Suppose we monitor the execution of $\mathcal{P}'$ on x. Then, we obtain a clustering Cl of G_x (each iteration of a loop body defines a separate cluster). We say that $\mathcal{P}'$ *corresponds* to the clustering Cl of G_x.

Lemma 2 (Transfer Lemma). *The following correspondences hold for $\mathcal{P}$ and inputs x of size n:*

- *If $\mathcal{P}' = [\![\mathcal{P}]\!](n)$ then $\mathcal{P}'$ corresponds to the naive transformation of G_x and $TIME_{\mathcal{P}'}(x) \leq (1 + 1/\gamma_{L,o,g}(G_x))T(G_x)$.*
- *If $\mathcal{P}' = [\![\mathcal{P}]\!]_L(n)$ then $\mathcal{P}'$ corresponds to a linear clustering of G_x and $TIME_{\mathcal{P}'}(x) \leq (1 + 1/\gamma_{L,o,g}(G_x))T(G_x)$.*
- *If $\mathcal{P}' = [\![\mathcal{P}]\!]_B(n)$ then $\mathcal{P}'$ corresponds to a Brent-clustering of G_x and $TIME_{\mathcal{P}'}(x) \leq (1 + 1/\gamma_{L,o,g}(G_x)) \times (W(G_x)/P + T(G_x))$.*

where the execution times to not count for the broadcasts.

We now add the broadcast times. Broadcasting on the LogP machine has been studied by [7] who showed that an greedy implementation (i.e. every processor sends as fast as possible the broadcast item as soon as it receives the item) is optimal. Provided that the processor initializing the broadcast is known in advance (as it is true in our case) all send and receive operations can be determined statically for an instance of the LogP machine, i.e. optimal broadcasting is oblivious. Let $B_{L,o,g}(P)$ be the execution time for a broadcast to P processors.

We relate now the execution time of a single loop iteration on the PRAM with the costs for broadcasting the termination condition to its output tasks. Let $\mathcal{P}$ be an iteratively oblivious PRAM-program. The *degree of obliviousness* of a loop l of $\mathcal{P}$ for inputs of size n, is defined by $\rho_{L,o,g}(l,n) = T(b)/(T(b) + B_{L,o,g}(n))$. where b is the body of l. Let $LOOPS(\mathcal{P})$ be the set of loops of $\mathcal{P}$ The *degree of obliviousness* of $\mathcal{P}$ is defined by $\rho_{L,o,g}(\mathcal{P},n) = \min_{l \in LOOPS(\mathcal{P})} \rho_{L,o,g}(l,n)$. The time bounds of Lemma 2 are delayed by at most a factor of $1/\rho_{L,o,g}(\mathcal{P},n)$. We define the notion of *granularity* of an iteratively oblivious PRAM-program $\mathcal{P}$ for all inputs of size n by $\gamma_{L,o,g}(\mathcal{P},n) = \min_{x \in I_n} \gamma_{L,o,g}(G_x)$ where I_n is the set of all inputs of size n. We obtain directly our final result:

Theorem 2 (Execution Time on Iteratively Oblivious Programs). *Let $\mathcal{P}$ be an iteratively oblivious PRAM-program. Then*

$$TIME_{[\![\mathcal{P}]\!]}(n) \leq \left(1 + \frac{1}{\gamma_{L,o,g}(\mathcal{P},n)}\right) \frac{T(\mathcal{P})}{\rho_{L,o,g}(\mathcal{P},n)}$$

$$TIME_{[\![\mathcal{P}]\!]_L}(n) \leq \left(1 + \frac{1}{\gamma_{L,o,g}(\mathcal{P},n)}\right) \frac{T(\mathcal{P})}{\rho_{L,o,g}(\mathcal{P},n)}$$

$$TIME_{[\![\mathcal{P}]\!]_B}(n) \leq \left(1 + \frac{1}{\gamma_{L,o,g}(\mathcal{P},n)}\right) \left(\frac{W(\mathcal{P})}{P} + T(\mathcal{P})\right) \rho_{L,o,g}(\mathcal{P},n)^{-1}$$

5 Conclusions

We extended the application of scheduling algorithms from task-graphs to iterative programs. The main idea is to apply standard scheduling algorithms (linear scheduling, Brent-scheduling) to the parts of the algorithm that can be modeled as task-graphs and to synchronize loop iterations by barrier synchronization. We showed that the well-known approximation factors for linear scheduling and Brent-scheduling for LogP-machines are divided by a *degree of obliviousness* $\rho_{L,o,g}$. This degree of obliviousness models how close an iterative algorithm comes to an oblivious algorithm (oblivious programs have a degree of obliviousness $\rho_{L,o,g} = 1$).

Although we can apply all scheduling algorithms for task graphs to iterative programs, we cannot generalize the above results on the time bounds to these algorithms. The problem occurs when the algorithm produces essentially different schedules for unrolled loops and composed loop bodies, i.e. the Transfer Lemma 2 is not satisfied.

The technique of loop synchronization might be useful for implementing scheduling algorithms in compilers. Instead of unrolling (oblivious) loops completely, it might be useful to unroll loops just a fixed number of times. In this case, we have the same situation as with iterative oblivious algorithms.

References

[1] 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.

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

[3] B. Di Martino and G. Ianello. Parallelization of non-simultaneous iterative methods for systems of linear equations. In *Parallel Processing: CONPAR 94 – VAPP VI*, volume 854 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 1994.

[4] Jörn Eisenbiegler, Welf Löwe, and Andreas Wehrenpfennig. On the optimization by redundancy using an extended LogP model. In *International Conference on Advances in Parallel and Distributed Computing (APDC'97)*, pages 149–155. IEEE Computer Society Press, 1997.

[5] I. Foster. *Design and Building Parallel Programs – Concepts and Tools for Parallel Software Engeneering.* Addison–Wesley, 1995.

[6] 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.

[7] 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.

[8] W. Lwe, J. Eisenbiegler, and W. Zimmermann. Optimizing parallel programs on machines with fast communication. In *9. International Conference on Parallel and Distributed Computing Systems*, pages 100–103, 1996.

[9] Welf Löwe, Wolf Zimmermann, and Jörn Eisenbiegler. On linear schedules for task graphs for generalized LogP-machines. In *Europar'97: Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 895–904, 1997.

[10] M. Philippsen. *Optimierungstechniken zur bersetzung paralleler Programmiersprachen.* PhD thesis, Universitt Karlsruhe, 1994. VDI-Verlag GmbH, Dsseldorf, 1994, VDI Fortschritt-Berichte 292, Reihe 10: Informatik.

[11] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison–Wesley, 1995.

[12] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computing.* ACM–Press, NY, 1990.

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