Parallel Processing Letters

ON OPTIMAL OROW-PRAM ALGORITHMS FOR COMPUTING RECURSIVELY DEFINED FUNCTIONS

ROLF NIEDERMEIER and PETER ROSSMANITH
Fakultät für Informatik, Technische Universität München,
Arcisstr. 21, D-80290 München, Fed. Rep. of Germany
Email: niedermr/rossmani@informatik.tu-muenchen.de

Received (received date)
Revised (revised date)
Communicated by (Editor Name)

ABSTRACT

We investigate parallel algorithms to compute recursively defined functions. Our computational model are parallel random access machines (PRAM's). We preferably make use of the OROW-PRAM (owner read, owner write), a model supposed to be even weaker and more realistic than the EREW-PRAM (exclusive read, exclusive write) and that still provides the opportunities of a completely connected processor network. For OROW-PRAM's we show that our parallel algorithms are work-optimal.

Keywords: optimal parallel algorithms, OROW-PRAM, recursively defined functions

1. Introduction

Many useful algorithms are recursive in structure. We investigate under which conditions recursively defined functions can be *directly* implemented on parallel computers. Our computational model is the *parallel random access machine* (PRAM for short) [4, 5, 6, 7]. A PRAM consists of an unbounded number of identical processors that work synchronously and that are connected via a global shared memory. PRAM's are classified according to their access mechanisms to shared memory. We restrict ourselves to the two in this respect weakest and, thus, most promising models for an efficient direct realization, that is, we consider the EREW-PRAM (exclusive read, exclusive write) [6, 7] and most of all the OROW-PRAM (owner read, owner write) [11]. In OROW-PRAM's each shared memory cell has a uniquely determined read-owner processor and a uniquely determined write-owner processor. These processors are the only ones allowed to read from and to write into a cell, respectively. Thus OROW-PRAM's may be viewed as a parallel computer where shared memory is replaced by communication links between pairs of processors.

Our main result is that for a lot of recursively defined functions under fairly natural conditions an optimal implementation on OROW-PRAM's is possible. More precisely, we demonstrate that for functions with cascade-type recursion work-optimal parallel algorithms exist. Herein the degree of parallelism only depends on the average width $\overline{w}$ of the tree of the recursive calls; we have optimal parallel algorithms using $O(\overline{w}/\log \overline{w})$ processors. The size of a recursion tree is a lower bound for the work of every parallel algorithm and the height of a recursion tree is a lower bound for the running time of every parallel algorithm. Thus the maximum number of processors that can be used in a work-optimal algorithm is $O(\overline{w})$ even if the model is arbitrarily powerful. So our OROW-PRAM algorithms with $O(\overline{w}/\log \overline{w})$ processors come quite close to the information-theoretic upper bound on the number of processors that holds for *every* parallel model. Work-optimal parallel algorithms for recursively defined functions may help to efficiently implement functional programming languages or might be useful in scientific computing.

Our solution strategy for the computation of recursively defined functions on OROW-PRAM's is as follows. First, note that for an efficient processor usage a load balancing strategy seems to be necessary. We solve this problem by giving a prefix sums algorithm working on the OROW-PRAM and by some well-suited allocation of read-owner and write-owner processors to shared memory cells. The basic techniques for all our algorithms are intimately related to the recursion tree corresponding to the computation of a recursive problem. Obviously, one can reduce the evaluation of a recursive function to the evaluation of this recursion tree in a straightforward way. There is also a related problem where we reduce the evaluation of a given arithmetic expression of length n to the evaluation of the corresponding expression tree [1, 2], which may be constructed bottom-up in a preprocessing phase. Abrahamson *et al.* [1] and Cole and Vishkin [2] give algorithms for EREW-PRAM's solving the expression evaluation problem in time $O(\log n)$ using $n/\log n$ processors, which is optimal. Their fundamental technique is called tree contraction, which heavily relies on the *static* structure of the expression tree. In the case of dealing with recursive functions and their recursion trees, the problem of evaluation becomes more delicate, since the recursion tree cannot be constructed in a preprocessing phase. This is due to its unknown size and structure that can only be ascertained dynamically during the evaluation. Storing the essential parts of the recursion tree in shared memory and working on it in parallel, we obtain our optimal algorithms that, moreover, generally need the same amount of memory space as the corresponding sequential algorithms. Is is noteworthy here that a completely connected machine like a PRAM seems to be necessary for a work-optimal solution. As Mayr and Werchner [9] pointed out, the tree contraction problem has no work-optimal solutions on the hypercube and, all the more, that holds for the more general problem of computing recursively defined functions. Thus OROW-PRAM's, which still model completely connected computers, provide an adequate framework.

To summarize, perhaps the two most important contributions of this paper are, on the one hand, dealing with an important but rather neglected subject like the computation of recursive functions on PRAM's and, on the other hand, the usage of a fairly realistic PRAM-type, that is, the OROW-PRAM.

The paper is organized as follows. In Section 2 we present some basic notation. In Section 3 we deal with the evaluation of recursive functions for the special case of associative and commutative functions on the EREW-PRAM and on the OROW-PRAM. In Section 4 we come to general cascade-type recursion. We conclude with some hints for the treatment of nested recursion and a discussion of the significance of our work.

2. Preliminaries

A *recursion tree* is the tree of the recursive calls that arise during the computation of a recursive function. Especially, the nodes of such a tree are labeled with the parameters of the recursive calls. If we speak of the generation of a recursion tree we say that a node becomes *expanded* or speak of an *expansion* of a node if we produce its children, i.e., carry out the recursive calls originating from this node. The size z and height h of a recursion tree are defined in the natural way. Clearly, the sequential time needed for the evaluation of a recursion tree is bounded from below by its size z. We call the set of nodes being at the same distance t from the root *level t nodes.* Finally, by w_t we understand the number of nodes at level t and $\overline{w}$ is defined as the average over all widths w_t of the recursion tree $(1 \leq t \leq h)$.

Dymond and Ruzzo [3] introduced the owner concept for the write access in PRAM's, yielding a frequently occurring subclass of the CREW-PRAM called CROW-PRAM. Following this line, that idea was also applied with respect to the read access of a PRAM, resulting in the OROW-PRAM model [11]. For an OROW-PRAM (owner read, owner write) there exist functions *write-owner* and *read-owner* that map each cell of shared memory to a unique processor that is the only one allowed to write into the cell or to read from the cell, respectively. A cell may have a write-owner distinct from its read-owner. Moreover, both owner functions have to be simple, i.e., computable in logarithmic space by a Turing machine. Apparently, an OROW-PRAM according to its communication possibilities is a PRAM model even weaker than the EREW-PRAM. The OROW-PRAM can be seen as a very weak but nevertheless natural member of the PRAM family, which has no shared memory but is provided with communication channels connecting each pair of processors.

Atomic operations of a PRAM are those which we assume to be computable in constant time. In the sequel we will give parallel algorithms that are *work-optimal*. The work done by the parallel algorithms, that is, the number of processors multiplied with the running time, is up to a constant factor the same as the work of the best sequential algorithm. For further details on PRAM's and related matters we also refer to the standard literature [4, 5, 6, 7].

Finally, let us shortly introduce the *prefix sums problem,* which is to play an important rôle in this work. Let $(M, \circ)$ be a semi-group. Given an array $X_1, \ldots, X_n$ of n elements from M, the problem is to determine the n prefix sums $S_i = X_1 \circ X_2 \cdots \circ X_i$, where $1 \leq i \leq n$. An optimal $O(\log n)$ time solution of the prefix sums problem on an EREW-PRAM using $n/\log n$ processors is well-known [8].

3. A Widespread Type of Recursion

In this section we concentrate on a widespread special case of cascade-type recursion. We show how to deal optimally with this form of recursion using EREW- and OROW-PRAM's. In the subsequent section we will build algorithms for more general and more intricate types of recursion upon the ideas presented here.

Proposition 1. *Let k be some constant and let $f : M \to M$ be a recursive function over a commutative monoid $(M, \circ, e)$ as follows:*

$$f(x) = \begin{cases} F(x) & \textit{if } C(x), \\ f(g_1(x)) \circ f(g_2(x)) \circ \cdots \circ f(g_k(x)) & \textit{otherwise.} \end{cases}$$

Herein, the functions $F, g_1, \ldots, g_k : M \to M$ and $C : M \to \{true, false\}$ are atomic. Then there exists an optimal EREW-PRAM algorithm computing f on input x using p processors for $p = O(\overline{w}/\log \overline{w})$.

Proof. The main data structure of our EREW-algorithm is an array A in shared memory that stores the current parameters of the recursive calls. The essential idea is to keep this array as compact and small as possible. The algorithm consists of three main parts. In the first part, *Spread,* we make available the memory cells in A for the parameters of the recursive calls. In the second part, *Compute,* we compute the parameters of the recursive calls in A or determine the termination of a recursion branch. Finally, in the third part, *Compact,* we partially compute the final result with the help of a variable S and remove the resulting gaps in A. Figure 1 contains an outline of the algorithm. Herein, A_i denotes the ith cell of array A in shared memory and B denotes an auxiliary global array for intermediate results. For the ease of presentation we assume that the recursive function f always terminates if $x = e$ (that is, $C(e) = true$). It only needs little effort to modify the algorithm if this is not the case. The integer variable w denotes the width of the recursion tree at the current leaf level and is initialized to 1 reflecting the fact that we begin at the root of the recursion tree. The variable S stores (partial) results in the computation of $f(x)$. The algorithm terminates if $w = 0$ since this means that all branches of the recursion tree have terminated and S contains $f(x)$. Let us only mention in pass that the current value of w may be distributed among the p processors in time $O(\log p)$ in the straightforward way.

It is obvious that an EREW-PRAM can perform the first two parts of the while-loop (*Spread* and *Compute*). The EREW property of the third part (*Compact*) follows from Ladner and Fischer's prefix sums algorithm [8], since it essentially consists of applications of prefix sums computations.

Optimality for the case $p = O(\overline{w}/\log \overline{w})$ is proved as follows. Clearly, in one iteration step *Spread* and *Compute* need time $O(w_t/p)$ using p processors ($1 \leq t \leq h$). For *Compact*, which essentially consists of prefix sums computations and the distribution of value w among p processors, the time complexity can be bounded from above by $O(\log w_t + w_t/p + \log p)$. Altogether, summing over all iterations of the algorithm (whose number is limited by h) and multiplying with the number of processors, we get a total work of $p \sum_{t=1}^{h} O(\log w_t + w_t/p + \log p)$. This simplifies

```
function f(x)
w ← 1; A_1 ← x; B_1 ← e; S ← e;
while w > 0 do begin
Spread:
  for i = 1 to w pardo begin
     (A_{k(i-1)+1}, A_{k(i-1)+2}, ..., A_{ki}) ← (A_i, e, ..., e);
  end
Compute:
  for i = 1 to w pardo begin
    if C(A_{k(i-1)+1}) then B_i ← F(A_{k(i-1)+1}); A_{k(i-1)+1} ← e;
    else (A_{k(i-1)+1}, ..., A_{ki}) ← (g_1(A_{k(i-1)+1}), ..., g_k(A_{k(i-1)+1})); B_i ← e;
  end
Compact:
  S ← S ∘ B_1 ∘ B_2 ∘ ··· ∘ B_w;
  A ← Concentrate(A);
  w ← Length(A);
end;
return S
```

Fig. 1. Algorithm for the associative and commutative case of cascade-type recursion: *Concentrate*(A) uses prefix sums computations to compact A by eliminating values e in A. *Length*(A) yields the number of elements different from e in A.

to $O(hp\log\overline{w} + h\overline{w} + hp\log p)$, since $\sum_{t=1}^{h}\log w_t \leq h\log\overline{w}$. For $p = O(\overline{w}/\log\overline{w})$ the work is $O(h\overline{w}) = O(z)$, which means work-optimality. □

Next, we show that the above algorithm also works for OROW-PRAM's. An optimal OROW-PRAM algorithm for prefix sums is necessary for that.

Lemma 1. *There is an OROW-PRAM algorithm for the prefix sums problem that runs in $O(\log n)$ time using $n/\log n$ processors, which is optimal. Herein we assume each processor to be read- and write-owner of $\log n$ consecutive cells of the input.*

Proof. The claim follows from the fact that the list ranking problem (which is more general than the prefix sums problem) for an input of length n can be solved by an OROW-PRAM in time $O(\log n)$ using n processors [11]. Observe that an input of length n can be reduced to one of length $n/\log n$ in time $O(\log n)$ using $n/\log n$ processors by performing local computations. □

It is straightforward to adapt Lemma 1 to the case when the number of processors p is smaller than $n/\log n$, resulting in a running time of $O(\max(\log n, n/p))$.

In general, the way how the input is assigned to the owner processors in Lemma 1 seems to be the most natural convention. But applying prefix sums with respect to some kind of load balancing for OROW-PRAM's, it is necessary to also have an algorithm where the ith processor is owner of input cells $i + lp$, where $0 \leq l \leq n/p - 1$. (Alternatively, one could make processor i owner of blocks of k consecutive cells of array A, i.e., of $(i-1)k + 1 + lpk, \ldots, ik + lpk$, where $0 \leq l \leq n/(pk) - 1$.) We transform this owner assignment to an instance suitable for application of Lemma 1. For this purpose, one has to use the *communication cell technique,* which is described in detail in the paper that introduced OROW-PRAM's [11]: Processor p_i can communicate with p_j if both of them know who the sender and the receiver of the information is. First, p_i writes into M_{ij}, then p_j reads from M_{ij}, where M_{ij} is a cell of shared memory with write-owner p_i and read-owner p_j. The communication cell technique is fundamental in the design of OROW-algorithms.

Now Proposition 1 can be stated in terms of OROW-PRAM's.

Corollary 1. *Let the assumptions for the recursive function f be the same as in Proposition 1. Then there exists an optimal OROW-PRAM algorithm computing f on input x using p processors for $p = O(\overline{w}/\log \overline{w})$.*

Proof. In essence, we employ the same algorithm as in Proposition 1. To equally distribute the work among p processors it suffices to lay down that processor i, where $1 \leq i \leq p$, is read- and write-owner for shared memory cells A_{i+lp} and B_{i+lp}. In this way we get a load balancing among the PRAM processors. It remains to be shown that the three parts *Spread, Compute,* and *Compact* (cf. Proposition 1 and Figure 1) can be implemented on an OROW-PRAM within the same complexity bounds as for an EREW-PRAM. For parts *Spread* and *Compute* this is trivial, since it is statically determined where to put the various data-items. For part *Compact* this immediately results from Lemma 1 and the remark before Corollary 1. □

4. More General Types of Recursion

In what follows our aim is to apply and generalize techniques of the previous section in order to attain the main result of this paper—an optimal parallel algorithm for cascade-type recursion.

We deal with the following form of recursion.

$$f(x) = \begin{cases} F(x) & \text{if } C(x), \\ H(x, f(g_1(x)), f(g_2(x)), \ldots, f(g_k(x))) & \text{otherwise.} \end{cases}$$

Herein the functions $C, F, H, g_1, \ldots, g_k$ are supposed to be atomic. The main difficulty in comparison with the special case of cascade-type recursion of the previous section is that we no longer can make use of an associative and commutative operation like there. Consequently, it is no longer possible to partially *compute* the final result $f(x)$ while working through the recursion tree T. This implies that it does not suffice to only store the values of the parameters corresponding to the current leaf level of the recursion tree, but one has to deal with the *whole* recursion tree up to the current leaf level. In particular, the value of the root of T can only be determined by starting at evaluated leaves and then "computing upwards" from the leaves to the root. By holding additional (pointer) information and making use of an evaluation strategy that consists of two phases, top-down from the root to the leaves (expansion of the tree) and bottom-up from the leaves to the root (evaluation of the recursion tree), nevertheless an optimal treatment of this kind of recursion by an OROW-PRAM is possible.

Before presenting our theorem for the above defined general cascade-type recursion, let us illustrate the essential concepts in a simple example.

Example 1. Consider the following simple, cascade-recursive function for non-negative integers x.

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \text{ or } x = 1, \\ x/(f(x-1)f(x-2)) & \text{otherwise.} \end{cases}$$

In Figure 2 we describe the expansion and contraction of the recursion tree for the computation of $f(4)$. Nodes enclosed in shaded areas are those on which the algorithm momentarily works (active nodes). Nodes whose values are already computed simply contain numbers and nodes which contain $f(.)$ are expanded in order to compute their value. So in Figure 2 we get seven stages of the partial recursion tree stored in shared memory during the execution of the algorithm.

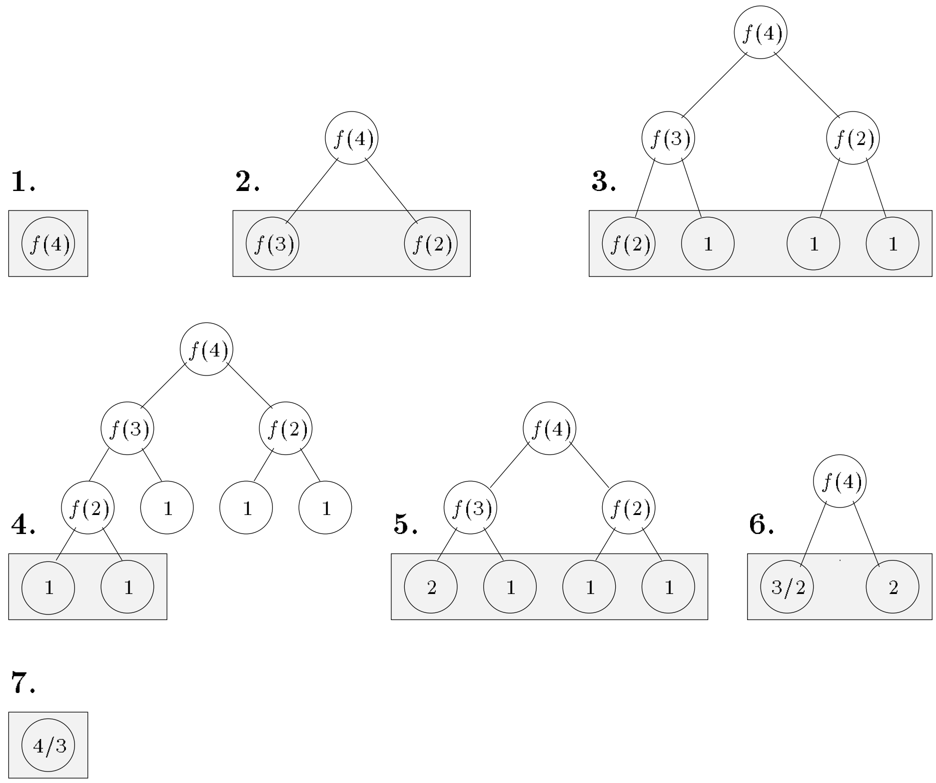

Fig. 2. Computation stages in the calculation of $f(4)$: 1. The initial recursive call. 2. Node $f(4)$ is expanded into $f(3)$ and $f(2)$. 3. Nodes $f(3)$ and $f(2)$ are expanded, where three of the resulting four children terminate with value 1. 4. Only the node $f(2)$ gets expanded. Now the complete recursion tree has been constructed and so the evaluation phase starts. 5. The value of node $f(2)$ is determined to be $2/(1 \cdot 1) = 2$. 6. The two nodes $f(3)$ and $f(2)$ are evaluated. 7. The value for the root is obtained, that is, $f(4) = 4/3$.

Figure 2 contains several important characteristics of the computation of cascade-type recursive functions. In particular, there is a clear separation between the phases of constructing the recursion tree (expansion phase) and of evaluating the recursion tree (contraction phase). No partial evaluations in the recursion tree take place (for example, in Figure 2 the right child $f(2)$ of $f(4)$ is not evaluated before the whole tree is constructed although it could have been done earlier in the computation), thus the expansion and the evaluation phase do not overlap. On the one hand, this will be of decisive importance in order to have an in some sense regular communication structure necessary for the implementation on an OROW-PRAM and, on the other hand, there is no increase of the worst case running time. So the basic idea in the following will be to assign processors only to the last two levels of the current partial recursion tree in order to enable an efficient and OROW-like handling of recursion.

Theorem 1. *Let f be a cascade-type recursive function. Then there exists an optimal OROW-PRAM algorithm computing f on input x using p processors for $p = O(\overline{w}/\log\overline{w})$.*

Proof. The rough structure of the algorithm is the same as in Proposition 1. Before we go into the details of the algorithm and its correctness, we start with a short outline. Note that we again have to make decisive use of the prefix sums computation on an OROW-PRAM provided by Lemma 1. The essential novelty in the following in comparison to Corollary 1 is that now we have first an expansion phase and second a contraction phase. These two phases do not overlap. Furthermore, processors always are assigned only to the two last levels of the recursion tree, one of them containing the current leaves and one of them containing their parents. This is necessary because of the limited communication possibilities of an OROW-PRAM. In the sequel we call these two levels the *active* (leaves) and the *pre-active* (parents) level, respectively. The active and pre-active levels will change after each expansion and after each contraction, as the following sketch of the algorithm shows.

The basic structure of the expansion phase is as follows. While there is at least one element in the active level that is to be expanded, do:

(i) Make the current active level the pre-active one.
(ii) Expand nodes in the pre-active level and store their children in memory cells of the new active level. In particular, generate both pointers from the parents to their children as well as pointers from the children to their parents.
(iii) Compact the new active level while maintaining the pointer information generated before. This is essential since because of termination not necessarily all nodes of the pre-active level had to be expanded.

When the expansion phase has finished its work, the contraction phase starts, having the following structure. As long as the root of the recursion tree has not been evaluated, perform the subsequent three statements:

(i) Compute the values of nodes of the current pre-active level by using the values stored in the active level, which have been computed in previous stages.
(ii) Make the current pre-active level the new active one.
(iii) Make the recursion tree level that contains the parents of the nodes belonging to the new active level the pre-active level.

Before beginning with a more detailed description of the expansion phase, we explain how to realize the use of an active and a pre-active level as described above.

Consider the part of shared memory storing the recursion tree as a two-dimensional array A. If we view A to consist of rows and columns, then we may say that each row of A contains a particular level of the recursion tree. To be more precise, the first row contains the root of the recursion tree, the second row the children of the root, the third row the grandchildren of the root and so on. In analogy to Corollary 1, in each row of array A processor i $(1 \leq i \leq p)$ is read- and write-owner of all cells with index $i + lp$. Thus a shared memory cell $A[j, m]$ contains the mth node in the jth level of the recursion tree together with additional pointer information. In this way, whether a row j is the current active or pre-active level may simply be determined using a counter for the current depth of the recursion tree, which can be held locally by each processor.

Now we go into the details of the expansion phase. In general, as in Proposition 1, this phase again basically consists of the three parts *Spread, Compute,* and *Compact.* The main difficulty in comparison to there is that now it is also necessary to take care of the use and maintenance of pointer information.

We restrict ourselves to pointing out the realization of the most difficult part of the expansion phase, i.e., part *Compact.* Remember that the purpose of *Compact* is to keep all levels or, equivalently, rows of memory storing the recursion tree as compact as possible.This is indispensable because due to termination some nodes of the recursion may not be expanded, although cells for their children were reserved during part *Spread.* The pointers from the children to their parents clearly will not change during the compaction of the active level. Therefore, we only have to consider the handling of the pointers from the parent nodes to their children. This is achieved in the following four steps.

(i) By means of a prefix sums computation determine the future address of a node in the active level.
(ii) The above newly computed address of a node is communicated to its parent node. This can be done by an OROW-PRAM making mainly use of the communication cell technique. Observe that as a rule processors always are assigned to cells of the active level *and* the pre-active level, enabling the use of the communication cell technique.
(iii) Now the processors assigned to the parent nodes update the pointers to their children, i.e., now the parents point to the positions where the children will be moved to in the next step. Thus, due to the static assignment of processors to memory cells in each row of A, the processors can now also compute the owners of the cells their children are moved to.
(iv) Eventually compact the active level by making use of a prefix sums computation in the same way as in Proposition 1. By this the children are moved to their (previously computed) new positions in the active level. After that the new width of the active level is computed and distributed among the p processors in a straightforward way.

Considerations analogous to the above ones also yield the realizability of the contraction phase. The correctness of the given algorithm follows from the fact that it directly evaluates the given recursion tree. Again, a simple but messy analysis (analogous to Proposition 1) yields the stated complexity bounds. In particular, observe that the algorithms of Corollary 1 and the above new one have the same worst case behavior in the sense that, e.g., a complete binary recursion tree in both cases requires a complete expansion and then contraction. □

In the above algorithm we constructed the whole recursion tree in shared memory in order to deal with general cascade-type recursion. In Proposition 1 and Corollary 1 we could avoid the construction of the complete binary tree while only dealing with the current leaf level. Clearly, Corollary 1 is a special case of Theorem 1, but nevertheless has a worth on its own because of the simpler algorithm and the smaller memory requirements. All our OROW-algorithms need shared memory in fact only for the communication cell technique. The parts of the recursion tree may be seen as being distributed among the local memories of the processors.

5. Conclusion

In this paper we confined our attention to cascade-type recursion. The basic ideas also apply to the treatment of recursive functions with nested recursive calls, although the details still get more complicated. It needs some care to define properly height and average width of recursion trees with nested recursive calls. Having done that, the main new technique to apply compared to Theorem 1 is to use a counter at each node of the recursion tree that provides information about the current "nestedness" of the node. Some more details may be found in an old version of this paper [10].

Let us shortly discuss the significance of our results. We provide optimal parallel algorithms for recursively defined functions if the number of processors is $O(\overline{w}/\log\overline{w})$, nearly meeting the information-theoretic upper bound of $O(\overline{w})$. It is of central importance to realize that our results hold for PRAM's with very restricted communication capabilities, i.e., the OROW-PRAM model. This gives further evidence for the fact that although an OROW-PRAM only possesses communication channels between pairs of processors, it nevertheless provides enough power for the efficient solution of important problems. Since an OROW-PRAM on the one hand still provides the opportunities of a completely connected processor network and, on the other hand, is the weakest and most realistic member of the PRAM family, it should become a standard model in design and analysis of parallel algorithms.

Acknowledgement

This work was supported by DFG-SFB 342 TP A4 "KLARA." We are grateful to Jörg Keller for several insightful remarks concerning the presentation of the work.

References

1. K. Abrahamson, N. Dadoun, D. A. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10:287–302, 1989.

2. R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.
3. P. W. Dymond and W. L. Ruzzo. Parallel RAMs with owned global memory and deterministic language recognition. In *Proc. of 13th International Conference on Automata, Languages, and Programming*, number 226 in Lecture Notes in Computer Science, pages 95–104. Springer, 1986.
4. A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
5. A. Gibbons and P. Spirakis, editors. *Lectures on parallel computation*. Cambridge International Series on Parallel Computation. Cambridge University Press, 1993.
6. J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
7. R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, chapter 17, pages 869–932. Elsevier, 1990.
8. R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27:831–838, 1980.
9. E. W. Mayr and R. Werchner. Optimal tree contraction on the hypercube and related networks. In T. Lengauer, editor, *Proc. of 1st European Symposium on Algorithms*, number 726 in Lecture Notes in Computer Science, pages 295–305, Bad Honnef, Federal Republic of Germany, Sept. 1993. Springer.
10. R. Niedermeier and P. Rossmanith. Optimal parallel algorithms for computing recursively defined functions. Technical Report TUM-I9218, SFB-Bericht Nr. 342/12/92 A, Technische Universität München, June 1992.
11. P. Rossmanith. The owner concept for PRAMs. In C. Choffrut and M. Jantzen, editors, *Proc. of 8th Symposium on Theoretical Aspects of Computer Science*, number 480 in Lecture Notes in Computer Science, pages 172–183, Hamburg, Federal Republic of Germany, Feb. 1991. Springer.