Optimal Parallel Recognition of Bracket Languages on Hypercubes

Gisela Pitsch Elmar Schömer
Lehrstuhl Prof. G. Hotz
Fachbereich 14 – Informatik
Universität des Saarlandes
D-6600 Saarbrücken, West Germany

Abstract

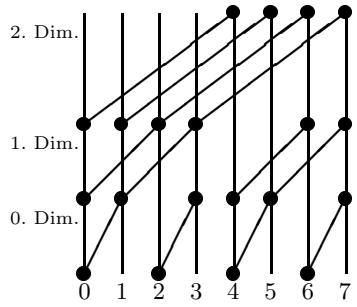
Bracket languages play an important role in the syntax analysis of programming languages. We investigate the parallel recognition and analysis of these languages as a first step towards a parallel working compiler. The main result consists in the design of an appropriate algorithm, which can be executed on hypercubes as well as on related bounded degree networks. In the average case we can achieve an optimal speed-up, meaning that q processors together can analyse bracket words of length N in time $O(N/q)$, if we restrict ourselves to employing no more than $\sqrt{N}$ processors.

1 Introduction

The basis for most programming languages are Context–Free languages. It is well known that CFL's can be recognized in polylogarithmic time using a polynomial number of processors [Rei], but these results are far from being optimal with respect to the speed-up.
Bracket languages as an important subclass of CFL's are also studied in the literature. So far the unique parallel algorithms for the recognition of bracket languages have been developed by Bar-On/Vishkin [BaVi], Rytter/Diks [RyDi] and Rytter/Giancarlo [RyGi]. They use PRAM's as their model of parallel computation. In contrast to them our algorithms are well suited for an implementation on a hypercube and its derived fixed degree networks such as the Cube–Connected–Cycles [PreVu] or the Perfect–Shuffle computer [St].
Our first algorithm uses a parallel sorting subroutine, whereas the second one is based on the paradigm of "divide & conquer". In the worst case their computation times only differ by logarithmic factors from the optimum. But the most remarkable thing about these efficient algorithms is that they exhibit an optimal behaviour in the average case: The theory of Random Walks enables us to give a good characterization of the structure of bracket words in the average case. By that means the design of the algorithms was guided. It goes without saying that parallelism can only be exploited up to a certain degree. The analysis of an input of length N can be accomplished in average time $O(N/q)$, but no more than $q = \sqrt{N}$ processors are permitted.

2 Basics

Our model of parallel computation is the hypercube. Although it is less powerful than the PRAM models it does not lack a certain universal character. Throughout the discussion, we let C^n denote the n–dimensional hypercube with $q = 2^n$ processors, represented by the set of all binary numbers in $\{0,1\}^n$. The $n2^{n-1}$ communication lines of C^n connect processors with identity numbers differing in precisely one bit. The dimension of a line is the bit position (0 to $n-1$) in which the incident processors differ.
In the plane this interconnection pattern looks as follows:



C^i denotes an i-dimensional subcube of the $\log q$-cube ($0 \le i \le \log q$) consisting of 2^i processors, whose id's are successively enumerated from $j \cdot 2^i$ to $\big((j+1)\cdot 2^i - 1\big)$ for any j, $0 \le j < 2^{\log q - i}$. Each such C^i can be divided into two subcubes of dimension $(i-1)$. Let $C^i := L^{(i-1)}R^{(i-1)}$. $L^{(i-1)}$ covers those processors of C^i, where the bit $(i-1)$ equals 0 (in the plane these ones represent the left half of all processors), whereas $R^{(i-1)}$ stands for the other (right) processors.
X represents a finite set of opening brackets and $\overline{X}$ the corresponding set of closing brackets with $X \cap \overline{X} = \emptyset$.

A pair $(x, \overline{x}) \in X \times \overline{X}$ is corresponding, if the brackets are of the same kind. This can be represented as a relation $\tau \subseteq X \times \overline{X}$ with $|\tau(x)| = 1$ for all $x \in X$.

We investigate words $w \in \left(X \cup \overline{X}\right)^*$. If two brackets forming a pair are directly neighboured, they may be cancelled. This corresponds to the calculation of the residual classes of $\left(X \cup \overline{X}\right)^*$ modulo τ. A sequence of brackets w is correctly nested, if and only if $w/\tau = \epsilon$. We call the set of all $w \in \left(X \cup \overline{X}\right)^*$ satisfying this condition D (according to the Dycklanguage [HoEs]).

In order to represent the input sequence of brackets $w \in \left(X \cup \overline{X}\right)^*$, we use the function *depth*. It associates with each bracket w_j $(0 < j \leq |w|)$ the depth of the nesting at position j of the input. Drawing the image of this function yields a kind of mountain as in figure 1. Let $w[p]$ be the sequence of brackets the processor with id p contains. Then $w(C^i) := w[j \cdot 2^i] \cdot w[j \cdot 2^i + 1] \cdot \ldots \cdot w[(j+1) \cdot 2^i - 1]$ for any j, $0 \leq j < 2^{\log q - i}$. Consequently, $|w(C^i)|$ denotes the length of the portion governed by all processors of C^i together.

Our aim is now to determine whether a given $w \in \left(X \cup \overline{X}\right)^*$ represents a word of the Dycklanguage D.

3 First Algorithm

The customary sequential method for the recognition of the Dycklanguage uses a stack. During the analysis of the word from left to right an opening bracket is put onto this stack and a closing one is tested whether it matches the bracket on top of the stack. In this case a cancellation takes place. Otherwise the input is rejected. The depth of the stack corresponds to our function *depth*. The depth of a Dyckword starts and ends at level 0 and in between it must not sink below this value.

The other well known method is the recognition by a Two–Way–One–Counter Automaton [HoMe]. It provides us with the key idea for a parallel approach to the problem: The corresponding counterpart to an opening bracket is the first bracket at the same *depth*–level, which succeeds this bracket.

We can get all the brackets at the same level of nesting within the input by drawing a parallel to the abscissa at this distance. The intersection points between the function and this parallel are related to these brackets. Figure 1 shows the graph of the *depth*–function for a correctly nested word and all matching pairs at level 2. Our aim is to permute the brackets of the input, so that corresponding brackets will be neighbouring inside the same processor. Then a final local test will suffice to solve the problem.

The solution is now evident: we sort the brackets w_j $(0 < j \leq |w|)$ according to the key $(depth(w, j), j)$. The resulting situation is as we wanted it to be if each processor obtains an even number of brackets.

Let the brackets of the input $w \in \left(X \cup \overline{X}\right)^*$, $|w| = N$, all be uniformly distributed over the $q \in \mathbf{N}$ processors available. Then we get the following:

ALGORITHM1:

(1) calculate $depth(w, j)$ for all $0 < j \leq |w|$

(2) sort the brackets w_j according to the key $(depth(w, j), j)$, $0 < j \leq |w|$, in increasing order

(3) run over the bracket sequence local in a processor while comparing brackets immediately neighboured

(4) tell everyone the result of all the comparisons

We now look at the single steps in detail.

Realisation

Let L be the maximal length of a word per processor.

Step(1) is based on the procedure PARALLEL_PREFIX [FiLa] and local calculations. One has to calculate all initial sums $S_j = \sum_{i=1}^{j} s_i \quad 1 \leq j \leq |w|$, where $s_i = +1$ for an opening bracket and -1 for a closing one. All this takes a time of $O(\log q + L)$.

Step(2) uses the procedure COLUMNSORT by Leighton [Lei]. It was chosen because it renders it possible to sort in optimal time, if only a small number of processors is given. Its implementation on the $\log q$-cube is straightforward because it only needs two different actions:

(a) sorting of an array internal to each processor, which costs $O(L \cdot \log L)$ operations, and

(b) exchange of information via the network; there are just two fixed permutations necessary corresponding to a shift and a transposition of a matrix; they all need no more than $O(L \cdot \log q)$ time.

In order to guarantee correctness, we are only allowed to put $q \leq \sqrt{L/2} + 1$ processors into action [Lei].

Step(3) does not need more explanations. Its computation time is $O(L)$.

The result of all the comparisons is contained in a boolean variable which is true if and only if the correspondences are all right or not.

Step(4) can be realized by using PARALLEL_PREFIX, where each initial sum corresponds to the boolean **and** of the results from step (3). This procedure costs $O(\log q)$ time.

Time complexity in the worst case

The time needed by ALGORITHM1 depends on the maximal length of a word per processor. Since $L \leq \lceil N/q \rceil$, the total running time of ALGORITHM1 comes to $O(N/q \cdot \log q)$ by using $q \leq \sqrt[3]{N/2}$ processors on a synchronous $\log q$-cube.

The restrictive processor bound resulting from COLUMNSORT can be further extended by using the more sophisticated sorting procedure CUBESORT [CySa] so that parallelism can be better exploited. We get the following

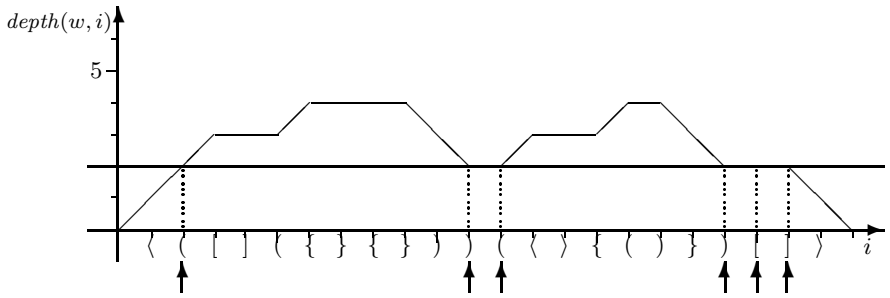


Figure 1: Brackets at the same level of nesting

Theorem 1 *The Dycklanguage D can be recognized in $O(N/q \cdot \log q)$ time by using $q < N^{1-\epsilon}$, $0 < \epsilon < 1$, processors connected as a $\log q$-cube. $N = |w|$ denotes the length of the input $w \in \left(X \cup \overline{X}\right)^*$.*

4 Second Algorithm

Another way of deciding the word problem for the Dycklanguage consists in the calculation of the reduced word. Later on it will become apparent that this method will be of a more general use than the method of the first algorithm.

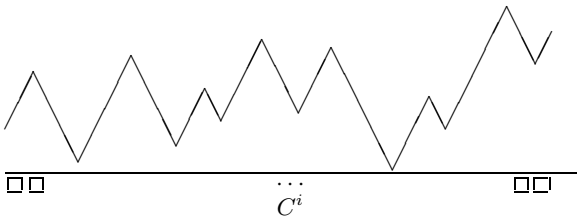


Figure 2: General structure of the algorithm's input

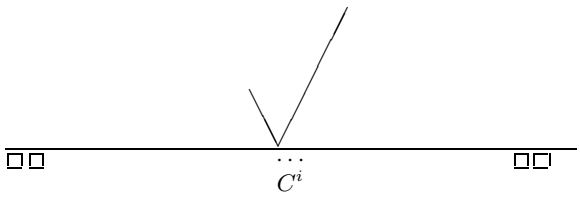


Figure 3: Reduced word

The brackets of the input are distributed uniformly over the processing elements available because their capacities should all be used. An input $w \in \left(X \cup \overline{X}\right)^*$ to the algorithm can now be illustrated as in figure 2. Therefrom it has to calculate the reduced word $w' \in \overline{X}^* X^*$ as shown in figure 3. This means: it has to remove all the peaks contained in the mountain. Hereby each peak corresponds to a certain number of opening brackets and their closing counterparts. The correctness of the procedure is derived from:

Lemma 1 [HoEs]
Let $u \in D$. It holds:

$$v_1 u v_2 \in D \iff v_1 v_2 \in D$$

Every peak removed corresponds to such an u.

The **idea** of the algorithm is based on the paradigm of "divide & conquer". In order to understand its application, let us look again at the function *depth*:

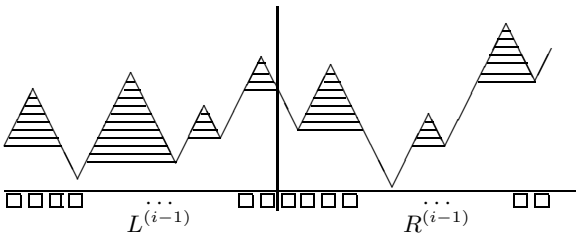


Figure 4: Situation given

Figure 4 shows a situation in a subcube C^i, where a sequence of brackets has been divided into two portions of equal length. The problem is solved for both in parallel by **recursively** removing the peaks local to the respective portion (see figure 5). There are 2^i processors. Half of them are working together on one portion by calculating the reduced word of their portion separately.
The remaining brackets have to be distributed again on the processors available. The fact that each one now gets a smaller number of brackets is represented by a smoother slope in figure 6. In addition the slope has to be the same on both sides of the separation in order to allow a correct cancellation. This means that processors at the same distance from the separation line on both sides have to own the same number of brackets. Let γ_{C^i} denote this number for the considered subcube C^i. The portion where the most brackets remain after a cancellation determines this number.

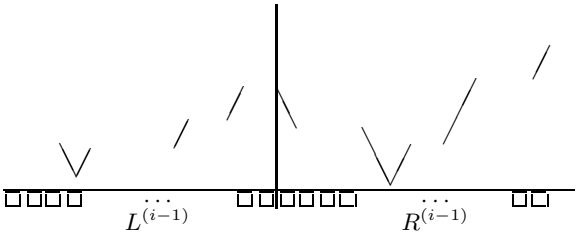


Figure 5: Removing the peaks

$$\gamma_{C^i} := \left\lceil \frac{1}{2^{i-1}} \cdot \max\left\{|w(L^{(i-1)})|, |w(R^{(i-1)})|\right\} \right\rceil$$

In general the first processors of L^{i-1} or the last processors of R^{i-1} will not get the full quantity. But we should not get confused with these subtleties here. A detailed description can be found in [Pi].

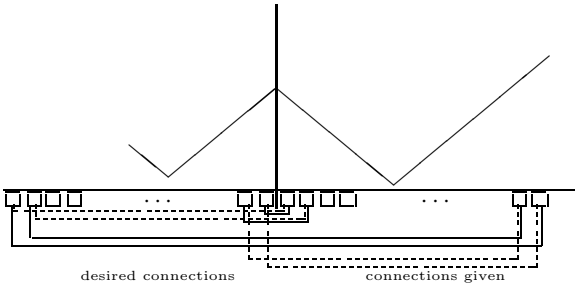


Figure 6: Situation after a distribution

Now we want to cancel again. In doing so we have to take the structure of the network into consideration, because the test as to whether two brackets form a pair or not can only be executed via these connections. We would like to have them between processors situated at the same distance from the separation line on both sides. But there are only connections in the $(i-1)$-th dimension leading from a processor $p \in L^{(i-1)}$ to the processor with the id $(p + 2^{(i-1)}) \in R^{(i-1)}$.
The solution consists in a reversion of one of the two portions. Now the cancellation is easy via the existing connections, as we can see in figure 7.

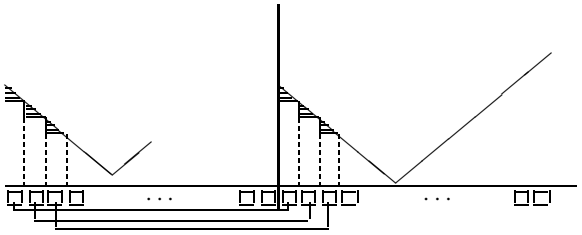


Figure 7: Obvious cancellation

The reversion itself can directly be realized on the hypercube by using its recursive definition:

Definition 1 *Let $w \in \left(X \cup \overline{X}\right)^*$. The* **reversion** *$w^R$ of w is formally defined by*

$$w^R := \begin{cases} w & \text{if } |w| = 1 \\ v^R u^R & w = uv; u, v \neq \epsilon \end{cases}$$

Figure 8 shows the information stream on the network. It is obvious that the time needed is proportional to the number of dimensions the brackets have to go through **times** the maximal number of brackets per processor during the step, because they all have to go through the connection successively.

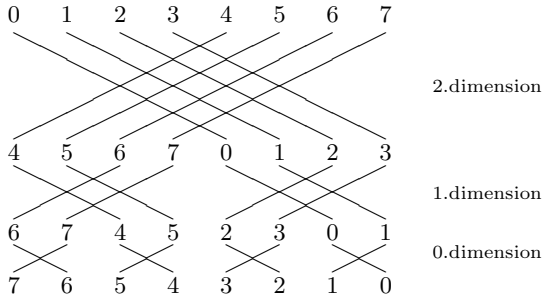


Figure 8: Reversion

Subsequent to the cancellation we have to restore the order of the brackets. This can be done by one more reversion since this is an operation inverse to itself. For the result see figure 9.

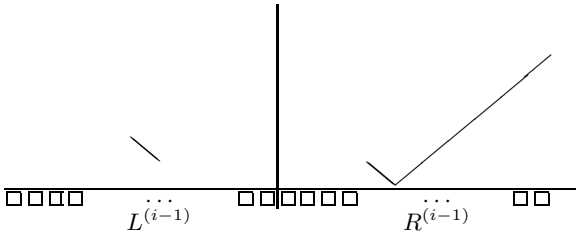


Figure 9: Result of the i-th recursion

With one more distribution we get the situation demanded at the beginning. The distribution itself is achieved by the following procedure:

(a) calculate the id of the processor each bracket has to reach

(b) route the brackets to these processors via the interconnection network

By summing up all observations made so far we get the following recursive procedure which calculates the reduced word corresponding to the input of a subcube. The symbol "‖" between two operations will signal that these two actions are done in parallel.

ALGORITHM2

procedure *reduce* $w(C^i)$
begin
if $i = 0$
then
(1) use the sequential algorithm
else
/* let $C^i := L^{(i-1)}R^{(i-1)}$ */
(2) *reduce* $w(L^{(i-1)})$ $\parallel$ *reduce* $w(R^{(i-1)})$
/* result: $w(L^{(i-1)}), w(R^{(i-1)}) \in \overline{X}^* X^*$ */
(3) *calculate* $|w(L^{(i-1)})|$ $\parallel$ *calculate* $|w(R^{(i-1)})|$
(4) *determine*
$\gamma_{C^i} := \left\lceil \frac{1}{2^{i-1}} \cdot \max\left\{|w(L^{(i-1)})|, |w(R^{(i-1)})|\right\}\right\rceil$
(5) *distribute* $w(L^{(i-1)})$ $\parallel$ *distribute* $w(R^{(i-1)})$
so that $\forall p \in L^{(i-1)}, R^{(i-1)}\ |w[p]| = \gamma_{C^i}$
(6) *reverse* $w(L^{(i-1)})$
(7) *cancel* corresponding brackets via the connections of dimension $(i-1)$ between $L^{(i-1)}$ and $R^{(i-1)}$
(8) *reverse* $w(L^{(i-1)})$
fi;
end;

The whole algorithm for a $\log q$-cube is started by "*reduce* $w(C^{\log q})$".

Realisation

We now have a closer look at each single step of the algorithm.
Let l_i be the maximal amount of brackets a processor contains during the i-th step of the recursion and $T(i)$ be the time needed by a call *reduce* $w(C^i)$. It consists of the following components:
Step(1) obviously needs $O(l_0) = O(N/q)$ operations.

Step(2) is a recursive call of the procedure. That's why it takes time $T(i-1)$.

Step(3) uses PARALLEL_PREFIX to determine the number of the brackets which are still remaining inside the 2^{i-1} processors working together. Therefore the total time needed here is $O(i)$.

Step(4) calculates the number of brackets each processor has to get as was described before. It costs constant time.

Step(5)**:**

(a) By knowing the results of step (3) and (4) the calculation of the destination processor for each bracket can be done by local operations in $O(l_i)$ time.

(b) In the case $N = q$, the routing corresponds to a concentration process of the brackets. That is why all the communication requests are of a special kind. If each bracket is moving to its destination along the shortest path, it is possible to show that there are no traffic jams. This was done by [NaSa] using $O(i)$ operations on an i-dimensional cube.
We generalize their procedure for smaller $q \in \mathbf{N}$ and achieve a running time of $O(i \cdot l_i)$ at the i-th step of the recursion. Supplementary it is necessary to show there that during the routing the maximal number of brackets per processor increases at most by a constant factor. This is proved in [Pi].

Steps(6) **and** (8) are evident according to the explanation before. They both cost $O(i \cdot l_i)$.

Step(7) consists of an examination of every single bracket contained in a processor, what can be done by $O(l_i)$ operations.

By summing up the running-time over all levels of recursion we obtain

$$T_{total}(N,q) = O\left(\frac{N}{q}\right) + \sum_{i=1}^{\log q - 1} O(i \cdot l_i)$$

Time-Complexity in the worst case

At the beginning each processor gets at most $\lceil N/q \rceil$ brackets. Even if there is not any cancellation possible up to the last recursion, the bound for l_i remains $\leq \lceil N/q \rceil$ for all $0 \leq i < \log q$. The use of this estimation results in the total time needed by our algorithm of

$$T_{total}(N,q) = O\left(\frac{N}{q} \cdot \log^2 q\right)$$

At first glance this algorithm seems to be computationally inferior to the first one. An average case study however will disprove this conjecture. But let us first look at the first algorithm in order to demonstrate how the analysis works.

5 The average case

The speed-up of $\frac{q}{\log q}$ in the first algorithm can be improved by a factor of $\log q$, if all the processors carry out a simple preprocessing:
They first eliminate those sequences illegally nested by the calculation of the function *depth*. Afterwards each one determines the reduced form of its input separately. All this costs only $O(\log q + \frac{N}{q})$ additional operations. It will be shown in the sequel that this simple strategy shortens the length N of the original input to $q \cdot \sqrt{N}$ in the average case. Due to this drastic reduction we can now afford to apply an expensive sorting procedure. If we use for example BITONIC SORT we can achieve $O(\sqrt{N} \log^2 q)$ sorting time. If we examine these costs together with the preprocessing, we see that we can accelerate our algorithm by the optimal factor. But we must restrict ourselves to employing at most $\frac{\sqrt{N}}{\log^2 N}$ processors.
Now we investigate why the local cancellation may cause such a significant reduction of the input length. Notice that inputs illegally nested needn't be considered anymore because they were eliminated before.

Let us have a closer look at the structure of the remaining words. Therefore, we use the function *depth* for a given w as in figure 10.
The y-coordinate of the highest peak of the image corresponds to the maximal depth D_{max} of the input w. We can draw a parallel to the x-axis at distance D_{max}. Every part of the function image is now lying in between the x-axis and this parallel.
It follows that the reduced words emerging from the preprocessing have a length at most $2 \cdot D_{max}$ independent of their previous lengths because all reduced words are elements of $\overline{X}^* X^*$.
As far as the maximal depth of a Dyckword is concerned, we know the following fact from the theory of Random Walks:

Theorem 2 [Ke]
Assuming that all Dyckwords $w \in \left(X \cup \overline{X}\right)^*$, $|w| = N$, are equally likely the average maximal depth of such a w is asymptotically given by

$$De(N) = \sqrt{\pi N/2} - \frac{1}{2} + O\left(\ln(N)/N^{\frac{1}{2}-\delta}\right)$$

for all $\delta > 0$.

The function given here corresponds to the expected value of the term

$$\max_{0<j\leq N} \{depth(w, j)\}$$

As it can easily be seen, this theorem is also true for any sequence of brackets where the nesting given by the function *depth* corresponds to that of a Dyckword. This means that the expected value of D_{max} is $De(N)$. Using the above reflections we can state as already mentioned that the length of the input reduces to $q \cdot \sqrt{N}$ in the average case.

Let us go on to the second algorithm. The reduction of the input at the beginning plays an important part there as well. But in addition the remaining brackets are distributed uniformly after each cancellation.
There are 2^i processors working together during the recursion $(i-1)$. And at the end of the recursion the brackets left ($\leq 2 \cdot D_{max}$) are distributed uniformly. The maximal amount of brackets each processor governs remains unchanged starting from this point until the end of the next recursion. Knowing all this we can conclude that during the i-th recursion for all $0 \leq i < \log q$ it holds:

$$l_i \leq \left\lceil \frac{2 \cdot D_{max}}{2^i} \right\rceil$$

Analogous with the above procedure we can identify D_{max} and $De(N)$.
Substituting l_i through $\left\lceil 2 \cdot De(N)/2^i \right\rceil$ in $T_{total}(N, q)$ results in

$$T_{total}(N, q) \leq O(\frac{N}{q} + \log^2 q + \sqrt{N})$$

This means that by using $q \leq \sqrt{N}$ processors we can achieve an optimal speed-up.
The smoother the slope of the function *depth* is, the faster the number of brackets per processor will be reduced and the earlier it will be possible to finish a step of the recursion. One possibility consists of running the network asynchronously. But it is also possible to determine global synchronisation points by calculating the maximal depth of the input first.

As result we get:

Theorem 3 *Let $w \in \left(X \cup \overline{X}\right)^*$, $|w| = N$, and let $q \leq \sqrt{N}$ processors be given. The question as to whether w represents a Dyckword can be decided in an average computation time of $O(N/q)$, when the processors are connected as a $\log q$-cube.*

There is still another fact known about the maximal depth of a $w \in D$:

Theorem 4 [Ke]
The variance of the maximal depth of a Dyckword with length N defined as a random variable is

$$\begin{aligned}\sigma^2(N) &= (\pi/3-1)\pi N/2 + \tfrac{1}{12} + \tfrac{5}{18}\pi^2 - \tfrac{11}{12}\pi \\ &\quad + O\left(\ln(N)/N^{\frac{1}{2}-\delta}\right)\end{aligned}$$

for all $\delta > 0$.

This can be used to calculate $Prob(|D_{max} - De(N)| \geq 2 \cdot De(N))$ for any given $w \in D$, $|w| = N$. This value is ≤ 0.06. The result is that the algorithms are optimal in at least 94% of all possible cases because a deviation of three times the expected value would not affect the average running time.
Their efficiency becomes even clearer, if we consider the fact that most of the nestings actually used are not very deep, e.g. in programming languages.

Conclusions

The first algorithm is conceptually simpler and in the worst case it only needs time $O(\frac{N}{q} \cdot \log q)$ on a $\log q$-cube, whereas the other one needs $O(\frac{N}{q} \cdot \log^2 q)$ there. But in order to obtain an optimal speed-up in the average case ALGORITHM2 can use more processors.
In addition the second algorithm determines the reduced form of inputs $w \in \left(X \cup \overline{X}\right)^*$, where $w \notin D$, and it can be generalized to solve the word problem for free groups. The necessary modifications confine to the cancellation process. In [Pi] it is shown that the worst case complexity is not affected. (In a free group we are allowed to cancel $x\overline{x}$ and $\overline{x}x$. This corresponds to the calculation of the residual classes relative to $(\tau \cup \tau^{-1})$.) After all, there is no problem to handle a modified cancellation relation with $\tau \subseteq X \times \overline{X}$ arbitrarily chosen.

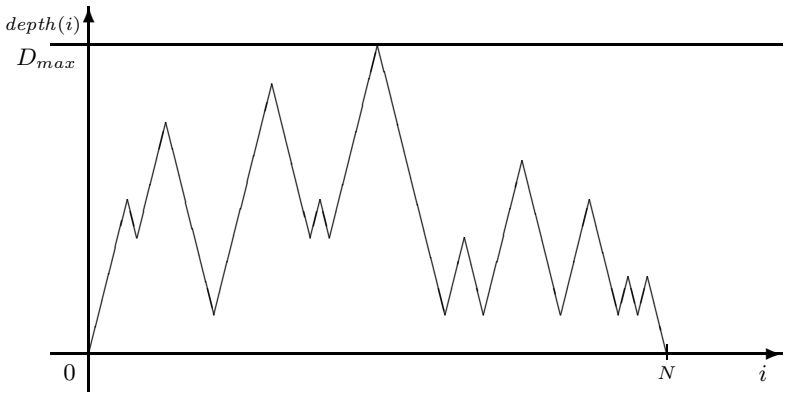


Figure 10: Maximal depth of a Dyckword

Acknowledgements

The authors gratefully acknowledge Prof. Dr. Günter Hotz for his comments and suggestions concerning this research. They also thank Prof. Dr. Bernd Becker, Thomas Burch, Andreas Nikolaus, Uwe Sparmann and Wolfgang Vogelgesang for a lot of helpful discussions.

References

[BaVi] I. Bar-On, U. Vishkin: *Optimal Parallel Generation of a Computation Tree Form.* ACM, Trans. Prog. Lang. and Syst. Vol. 7, No. 2, April 1985, pp. 348-357

[CySa] R. Cypher, J.L.C. Sanz: *Cubesort: An Optimal Sorting Algorithm for Feasible Parallel Computers.* LNCS 319, pp. 456-464

[FiLa] M.J. Fischer, R.E. Ladner: *Parallel Prefix Computation.* J. Ass. Comp. Mach., Vol. 27, 1980, pp. 839-849

[HoEs] G. Hotz, K. Estenfeld: *Formale Sprachen.* Bibliographisches Institut (1981)

[HoMe] G. Hotz, J. Messerschmidt: *Dycksprachen sind in Bandkomplexität* $\log n$ *analysierbar.* Techn. Rep. A75/1, Universität des Saarlandes, 1975

[Ke] R. Kemp: *Fundamentals of the Average Case Analysis of Particular Algorithms.* Wiley-Teubner (1984)

[Lei] T. Leighton: *Tight Bounds on the Complexity of Parallel Sorting.* IEEE Trans. on computers, Vol. C34, 4, April 1985, pp. 344-354

[NaSa] D. Nassimi, S. Sahni: *Data Broadcasting in SIMD Computers.* IEEE Trans. on computers, Vol. C30, 2, Feb. 1981, pp. 101-107

[Pi] G. Pitsch: *Effiziente parallele Verfahren zur Entscheidung des Wortproblems bei Dycksprachen.* Master's Thesis, Universität des Saarlandes, Saarbrücken, 1989

[PreVu] F. Preparata, J. Vuillemin: *The Cube-Connected Cycles: A versatile network for parallel computation.* 20^{th} FOCS (1979), pp. 140-147

[Rei] J. Reif: *Parallel time* $O(\log n)$ *acceptance of deterministic cfl's.* 23^{rd} FOCS (1982)

[RyDi] W. Rytter, K. Diks: *On optimal parallel computations for sequences of brackets.* Workshop "*Sequences*", Positano, June 1988

[RyGi] W. Rytter, R. Giancarlo: *Optimal parallel parsing of bracket languages.* Theoretical Computer Science 53 (1987), pp. 295-306

[St] H. S. Stone: Parallel processing with the perfect shuffle. IEEE Trans. on computers, Vol. C20, 2, February 1971, pp. 153-161