Parallel Processing Letters Vol. 9 No. 3 (1999) 335–345

CALCULATING AN OPTIMAL HOMOMORPHIC ALGORITHM FOR BRACKET MATCHING

ZHENJIANG HU* MASATO TAKEICHI
Department of Information Enginnering, University of Tokyo
Hongo 7-3-1, Bunkyo, Tokyo 113, Japan

Received April 1998
Revised July 1998
Accepted by C. Lengauer

ABSTRACT

It is widely recognized that a key problem of parallel computation is in the development of both efficient and correct parallel software. Although many advanced language features and compilation techniques have been proposed to alleviate the complexity of parallel programming, much effort is still required to develop parallelism in a formal and systematic way. In this paper, we intend to clarify this point by demonstrating a formal derivation of a correct but efficient homomorphic parallel algorithm for a simple language recognition problem known as bracket matching. To the best of our knowledge, our formal derivation leads to a novel divide-and-conquer parallel algorithm for bracket matching.

Keywords: Skeleton, Parallel Functional Programming, List Homomorphism, Parallelization, Bird-Meertens Formalisms, Program Calculation

1. Introduction

The bracket matching is a kind of language recognition problems, determining whether the brackets in a given string are correctly matched. For example, for the language generated by the grammar

$$\begin{array}{lcl} S & \rightarrow & S\,S \\ S & \rightarrow & (\,S\,) \\ S & \rightarrow & [\,S\,] \\ S & \rightarrow & \{\,S\,\} \\ S & \rightarrow & \text{any other symbol} \end{array}$$

the string "$g + \{[o + o] * d\}()$" is accepted, whereas "$b\{[a)d]$" is not. This problem is of interest in parallel programming in that the problem itself is so simple but finding an efficient parallel algorithm is far from being trivial.

It is known that this problem can be solved using $O(\log n)$ parallel time on $O(n/\log n)$ processors in the PRAM model [7], where n denotes the length of the

*Correspondence Address: Zhenjiang Hu, Information Processing Lab., Dept. of Information Engineering, Univ. of Tokyo, Tokyo 113 Japan. Tel. 81-3-3812-2111×7411. Email: hu@ipl.t.u-tokyo.ac.jp

input string. But the algorithms involved are rather complicated. To remedy this situation, Cole [6] investigated a methodology for the development of parallel algorithms based upon exploration of parallelism of homomorphisms in Bird Meertens Formalisms [2]. He successfully applied it to derive a homomorphic parallel algorithm for bracket matching. However, there still remain two major problems.

- The derivation proposed by Cole is quite informal, although much effort has been made to show intuitively the correctness of the designed algorithms. As concluded in [6], *It is of interest to ask how easily the resulting algorithm might have been derived in a more strictly formal setting.*
- The derived homomorphic algorithm is *not optimal*; it needs the parallel time of $O(\log^2 n)$ rather than the optimal $O(\log n)$ [7]. In a similar approach to Cole's, a homomorphic parsing algorithm [1] is presented for operator precedence grammars, from which a solution to the bracket matching problem can be specialized, but it has linear time behavior in the worst case. It would be more convincing if we could derive an optimal homomorphic algorithm.

This paper shows how these two problems can be well solved based on the parallelization theorem [15]. In particular, we will propose a systematic and formal derivation of, to the best of our knowledge, a *novel* optimal homomorphic algorithm for bracket matching.

2. List Homomorphisms

List homomorphisms, an important concept in Bird Meertens Formalisms (BMF for short) [2], play a central role in our derivation. To make the paper self-contained, we briefly explain some notational conventions of BMF that will be used later.

Function application is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried, and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Function application binds stronger than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. *Function composition* is denoted by a centralized circle $\circ$. By definition, we have $(f \circ g)\,a = f\,(g\,a)$. Function composition is an associative operator, and the identity function is denoted by *id*. *Infix binary operators* will often be denoted by $\oplus, \otimes$. *Lists* are finite sequences of values of the same type. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a, and $x +\!\!+ y$ for the concatenation of two lists x and y. The term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : xs$ for $[a] +\!\!+ xs$.

List homomorphisms become more and more attractive in parallel programming [2,6,9,10,11,12,15], mainly because of their distinguished properties of simplicity, clear parallelism, and manipulability.

- First, they are the *simplest* recursive skeletons on join lists; function h is a list homomorphism† if there exist a function k and an associative binary operator

† We assume that h is defined over nonempty lists in this paper.

$\oplus$ so that h is defined by

$$\begin{array}{lcl} h\ [a] & = & k\ a \\ h\ (x +\!\!+ y) & = & h\ x \oplus h\ y. \end{array}$$

For instance the function *sum* that sums up all elements of a list is a list homomorphism, in which $k = id$ and $\oplus = +$.

- Second, list homomorphisms attain a potentially high *parallelism*. Intuitively, the value of h on a larger list depends in a particular way (using binary operation $\oplus$) on the values of h applied to the two pieces of the list, which can be viewed as expressing the well-known divide-and-conquer paradigm in parallel programming.
- Third, list homomorphisms are *manipulable* (suitable for program transformation), because they enjoy many nice algebraic laws, such as the homomorphism lemmas [2,8], the fusion and tupling transformation laws [12], and the parallelization theorem [15].

3. The Parallelization Theorem

The following gives a parallelization lemma. It is a special case of the general parallelization theorem [15], but is suffice for us to calculate parallel algorithms for bracket matching.

Lemma 1 (Parallelization) Given is a program

$$\begin{array}{lcl} f\ [a]\ c & = & g_0\ a\ c \\ f\ (a : x)\ c & = & g_1\ a\ c\ \oplus\ f\ x\ (g_2\ a \otimes c) \end{array}$$

where $\oplus$ and $\otimes$ are two associative binary operators. Then, for any non-empty lists x' and x, we have

$$\begin{array}{lcl} f\ [a]\ c & = & g_0\ a\ c \\ f\ (x' +\!\!+ x)\ c & = & G_1\ x'\ c \oplus f\ x\ (G_2\ x' \otimes c) \end{array}$$

where G_1 and G_2 are functions defined by

$$\begin{array}{lcl} G_1\ [a]\ c & = & g_1\ a\ c \\ G_1\ (x'_1 +\!\!+ x'_2)\ c & = & G_1\ x'_1\ c\ \oplus\ G_1\ x'_2\ (G_2\ x'_1 \otimes c) \\ G_2\ [a] & = & g_2\ a \\ G_2\ (x'_1 +\!\!+ x'_2) & = & G_2\ x'_2 \otimes G_2\ x'_1 \end{array}$$

□

We will not recap the proof of the lemma as given in [15]. Instead, we demonstrate how it works for calculating parallel algorithms. We consider a simplified bracket matching problem: determining whether a single type (rather than many types) of brackets, '(' and ')', in a given string are correctly matched. This problem has a straightforward linear sequential algorithm, in which the string is examined

from left to right. A counter is initialized to 0, and increased or decreased as opening and closing brackets are encountered.

$$\begin{array}{lcl} sbp'\ [\,]\ c & = & c == 0 \\ sbp'\ (a : x)\ c & = & \text{if } a == \text{'('} \text{ then } sbp'\ x\ (c+1) \\ & & \text{else if } a == \text{')'} \text{ then } c > 0 \wedge sbp'\ x\ (c-1) \\ & & \text{else } sbp'\ x\ c. \end{array}$$

Unifying three occurrences of the recursive call into a single one using the Normalizing transformation of conditional structures [5] gives

$$\begin{array}{lcl} sbp'\ (a : x)\ c & = & (\text{if } a == \text{'('} \text{ then } \mathit{True} \\ & & \text{else if } a == \text{')'} \text{ then } c > 0 \text{ else } \mathit{True}) \\ & & \wedge \\ & & sbp'\ x\ ((\text{if } a == \text{'('} \text{ then } 1 \\ & & \text{else if } a == \text{')'} \text{ then } -1 \text{ else } 0) + c). \end{array}$$

Now we can apply the lemma by introducing two functions g_1 and g_2 to abstract two subexpressions

$$\begin{array}{lcl} sbp'\ (a : x)\ c & = & g_1\ a\ c \wedge sbp'\ x\ (g_2\ a + c) \\ g_1\ a\ c & = & \text{if } a == \text{'('} \text{ then } \mathit{True} \\ & & \text{else if } a == \text{')'} \text{ then } c > 0 \text{ else } \mathit{True} \\ g_2\ a & = & \text{if } a == \text{'('} \text{ then } 1 \\ & & \text{else if } a = \text{')'} \text{ then } -1 \text{ else } 0 \end{array}$$

and obtain

$$sbp'\ (x' +\!\!+ x)\ c = G_1\ x'\ c \wedge sbp'\ x\ (G_2\ x' + c)$$

where

$$\begin{array}{lcl} G_1\ [a]\ c & = & \text{if } a == \text{'('} \text{ then } \mathit{True} \\ & & \text{else if } a == \text{')'} \text{ then } c > 0 \\ & & \text{else } \mathit{True} \\ G_1\ (x_1' +\!\!+ x_2')\ c & = & G_1\ x_1'\ c \wedge G_1\ x_2'\ (G_2\ x_1' + c) \\ G_2\ [a] & = & \text{if } a == \text{'('} \text{ then } 1 \\ & & \text{else if } a == \text{')'} \text{ then } (-1) \text{ else } 0 \\ G_2\ (x_1' +\!\!+ x_2') & = & G_2\ x_2' + G_2\ x_1' \end{array}$$

This is the parallel version we aim to get in this paper, although it is currently inefficient because of multiple traversals of the same input list by several functions. But this can be automatically improved by the tupling calculation as intensively studied in [14]. For instance, we can obtain the following program by tupling sbp', G_1 and G_2.

$$\begin{array}{lcl} sbp'\ x\ c & = & s \text{ where } (s, g_1, g_2) = tup\ x\ c \\ tup\ [a]\ c & = & \text{if } a == \text{'('} \text{ then } (c+1 == 0, \mathit{True}, 1) \\ & & \text{else if } a == \text{')'} \text{ then } (c-1 == 0, c > 0, -1) \\ & & \text{else } (c == 0, \mathit{True}, 0) \\ tup\ (x +\!\!+ y)\ c & = & \text{let } (s_x, g_{1x}, g_{2x}) = tup\ x\ c \\ & & \quad (s_y, g_{1y}, g_{2y}) = tup\ y\ (g_{2x} + c) \\ & & \text{in } (g_{1x} \wedge s_y,\ g_{1x} \wedge g_{1y},\ g_{2y} + g_{2x}) \end{array}$$

It seems not so apparent that the above gives an efficient parallel program. Particularly, the second recursive call $tup\ y\ (g_{2x} + c)$ relies on g_{2x}, an output from the first recursive call $tup\ x\ c$. Nevertheless, this version of *tup* can be efficiently implemented in parallel on a multiple processor system supporting bidirectional tree-like communication with $O(\log n)$ complexity where n denotes the length of the input list, by using an algorithm similar to that in [4]. Two passes are employed; an upward pass in the computation is used to compute the third component of $tup\ x\ c$ before a downward pass is used to compute the first two values of the tuple.

To summarize the above, we can get the following performance lemma.

Lemma 2 (Performance) In Lemma 1, the parallelized f can be implemented using $O(max(t_{g_0}, t_{g_1}, t_{g_2}) + max(t_{\oplus}, t_{\otimes}) \times \log n)$ parallel time on $(n/\log n)$ processors in the PRAM model, where n denotes the length of the input list, and the notation t_{op} denotes the time for computing op. □

4. Derivation

We return to our main topic; deriving an efficient homomorphic parallel algorithm for bracket matching (of many types of brackets). We start with a naive sequential program, and then try to parallelize it using the parallelization lemma. The key to our derivation is a new idea for constructing an efficient associative operator to combine two stacks.

4.1. Specification

Compared to the bracket matching of a single type of brackets, arbitrary bracket types complicates the bracket matching problem. But a simple straightforward linear time sequential algorithm still exists by using a stack. Opening brackets are pushed, and a closing brackets are matched with the current stack top. Failure is indicated by a mismatch, or by a nonempty stack when a match is required or at the end of the scan of the input. Thus we come to the following straightforward specification.

$$
\begin{array}{lcl}
bm\ [\,]\ s & = & isEmpty\ s \\
bm\ (a : x)\ s & = & \text{if } isOpen\ a \text{ then } bm\ x\ (push\ a\ s) \\
 & & \text{elseif } isClose\ a \text{ then } noEmpty\ s\ \wedge\ match\ a\ (top\ s)\ \wedge \\
 & & \qquad\qquad bm\ x\ (pop\ s) \\
 & & \text{else } bm\ x\ s
\end{array}
$$

Here we use several boolean functions; *isOpen* and *isClose* are to determine if a symbol is an opening or an closing bracket, *match* to determine if two symbols are bracket-matched, e.g., *match* '[' ']' gives *True* whereas *match* '(' ']' gives *Flase*, and *isEmpty* and *noEmpty* to determine if a stack is empty or not.

4.2. Linearization

We intend to use the parallelization lemma to derive a parallel algorithm from the specification. Comparing the specification with the program that can be accepted

by the parallelization lemma indicates that three occurrences of the recursive call should be unified into a single one, and that we a definition of bm for the case of a singleton list should be given. The formal can be done in a similar way as for sbp', and the latter can be done by a simple in-lining.

$$\begin{array}{lcl} bm\ [a]\ s & = & g_0\ a\ s \\ bm\ (a:x)\ s & = & g_1\ a\ s\ \wedge\ bm\ x\ (g_2\ a\ s) \end{array}$$

where

$$\begin{array}{lcl} g_0\ a\ s & = & \text{if } isOpen\ a \text{ then } False \\ & & \text{elseif } isClose\ a \text{ then} \\ & & \quad noEmpty\ s\ \wedge\ match\ a\ (top\ s)\ \wedge\ isEmpty\ (pop\ s) \\ & & \text{else } isEmpty\ s \\ g_1\ a\ s & = & \text{if } isOpen\ a \text{ then } True \\ & & \text{elseif } isClose\ a \text{ then } noEmpty\ s\ \wedge\ match\ a\ (top\ s) \\ & & \text{else } True \\ g_2\ a\ s & = & \text{if } isOpen\ a \text{ then } push\ a\ s \\ & & \text{elseif } isClose\ a \text{ then } pop\ s \\ & & \text{else } s. \end{array}$$

4.3. Deriving an associative operator

Now the function bm is in an accepted form that the parallelization lemma can be applied, provided that there exists an associative operator $\otimes$ such that g_2 can be expressed by

$$g_2\ a\ s = g_2'\ a\ \otimes\ s.$$

Fortunately, we know that an associative operator can be systematically derived from many algebraic data types [16,15]. Let the stack be defined by [‡]

$$Stack\ =\ Empty\ |\ Push\ Char\ Stack\ |\ Pop\ Stack$$

from which we know that an associative operator $\otimes$ (also called *zero-replacement* function which inductively replaces zero-constructor $Empty$ by another stack) can be derived using the standard technique [16,15].

$$\begin{array}{lcl} Empty \otimes s & = & s \\ (Push\ a\ s') \otimes s & = & Push\ a\ (s' \otimes s) \\ (Pop\ s') \otimes s & = & Pop\ (s' \otimes s) \end{array}$$

It is left for reader to check that $\otimes$ is indeed associative and has $Empty$ as its identity unit. Consequently, we can extract s out of g_2 using $\otimes$ and obtain

$$\begin{array}{lcl} g_2\ a\ s & = & g_2'\ a \otimes\ s \\ g_2'\ a & = & \text{if } isOpen\ a \text{ then } push\ a\ Empty \\ & & \text{elseif } isClose\ a \text{ then } pop\ Empty \\ & & \text{else } Empty \end{array}$$

[‡]Notice that we include Pop as a data constructor that is usually considered as a destructor of the stack. By doing so, we can postpone constructing stack.

>From the stack property $Pop \circ Push\ a = id$, we can see that our stack can be kept in the following normal form

$$Push\ a_1\ (Push\ a_2\ (\cdots (Push\ a_n\ (Pop\ (Pop(\cdots (Pop\ Empty)))))))) \qquad (1)$$

that is, the occurrences of the *Pop* constructor, if there are, should appear inside the *Push* constructors, but not vice versa. With this normal form, we give definitions for those functions that manipulate the stack.

$$\begin{array}{lcl} push\ a\ s & = & Push\ a\ s \\ pop\ s & = & Pop\ s \\ top\ (Push\ a\ s) & = & a \\ isEmpty\ Empty & = & True \\ isEmpty\ \_ & = & Flase \\ noEmpty & = & not \circ isEmpty \end{array}$$

4.4. Efficiently Implementing $\otimes$

Applying the parallelization lemma will soon give our homomorphic algorithm. But according to the performance lemma we require that $\otimes$ be implemented in constant parallel time, in order to obtain an $O(\log n)$ parallel algorithm. As a matter of fact, the present definition is unsatisfactory; it is sequential. We shall show how to implement $\otimes$ efficiently in parallel.

Recall that our stack has the form of (1), which can be represented by a list of $a_1, \cdots, a_n$ and the number of uses of the *Pop* constructor, i.e.,

$$([a_1, \cdots, a_n], n, m)$$

where m denotes the number of *Pop*'s. Note that to simplify our later presentation, we include the second element n so that the length of the first component can be computed incrementally. With this representation, we can implement the stack constructors as follows.

$$\begin{array}{lcl} Empty & = & ([\,], 0, 0) \\ Push\ c\ (cs, n, m) & = & ([c] +\!\!+ cs, n+1, m) \\ Pop\ (c : cs, n+1, m) & = & (cs, n, m) \\ Pop\ ([\,], 0, m) & = & ([\,], 0, m+1) \end{array}$$

And

$$\begin{array}{l} (cs_1, n_1, m_1) \otimes (cs_2, n_2, m_2) \\ \quad = \text{if } m_1 \geq n_2 \text{ then } (cs_1, n_1, m_1 - n_2 + m_2) \\ \qquad \text{else } (cs_1 +\!\!+ drop\ m_1\ cs_2, n_1 + n_2 - m_1, m_2) \end{array}$$

where $drop\ n\ xs$ drops off the first n elements of xs. Because *drop* and $+\!\!+$ can be implemented in parallel using constant time, it follows that $\otimes$ can be implemented in constant parallel time.

4.5. Applying the parallelization lemma

$$
\begin{array}{lcl}
bracketMatching & :: & String \to Bool \\
bracketMatching\ x & = & \text{let } (bm, g_1, g_2) = tup\ x\ Empty \text{ in } bm \\
tup\ [a]\ s & = & \text{if } isOpen\ a \text{ then } (False, True, push\ a\ Empty) \\
 & & \text{elseif } isClose\ a \text{ then } (noEmpty\ s \wedge match\ a\ (top\ s) \wedge \\
 & & \qquad isEmpty\ (pop\ s), \\
 & & \qquad noEmpty\ s \wedge match\ a\ (top\ s), \\
 & & \qquad pop\ Empty) \\
 & & \text{else } (isEmpty\ s, True, Empty) \\
tup\ (x +\!\!+ y)\ s & = & \text{let } (bm_x, g_{1x}, g'_{2x}) = tup\ x\ s \\
 & & \quad (bm_y, g_{1y}, g'_{2y}) = tup\ y\ (g'_{2x} \otimes s) \\
 & & \text{in } (g_{1x} \wedge bm_y,\ g_{1x} \wedge g_{1y},\ g'_{2y} \otimes g'_{2x})
\end{array}
$$

where

$$
\begin{array}{lcl}
Empty & = & ([\,], 0, 0) \\
push\ a\ (x, n, m) & = & ([a] +\!\!+ x, n+1, m) \\
pop\ (a : x, n+1, m) & = & (x, n, m) \\
pop\ ([\,], 0, m) & = & ([\,], 0, m+1) \\
isEmpty\ s & = & \text{if } s == ([\,], 0, 0) \text{ then } True \text{ else } False \\
noEmpty\ s & = & not(isEmpty\ s)
\end{array}
$$

and

$$
\begin{array}{l}
(x_1, n_1, m_1) \otimes (x_2, n_2, m_2) \\
\quad = \text{if } m_1 \geq n_2 \text{ then } (x_1, n_1, m_1 - n_2 + m_2) \\
\qquad \text{else } (x_1 +\!\!+ drop\ m_1\ x_2, n_1 + n_2 - m_1, m_2)
\end{array}
$$

Figure 1: The Final Program for Bracket Matching

Now we are ready to apply the parallelization lemma, and obtain the following parallel algorithm for bracket matching.

$$
\begin{array}{lcl}
bm\ [a]\ s & = & g_0\ a\ s \\
bm\ (x' +\!\!+ x)\ s & = & G_1\ x'\ s\ \wedge\ bm\ x\ (G_2\ x' \otimes s) \\
G_1\ [a]\ s & = & g_1\ a\ s \\
G_1\ (x'_1 +\!\!+ x'_2)\ s & = & G_1\ x'_1\ s \wedge G_1\ x'_2\ (G'_2\ x'_1 \otimes s) \\
G_2\ [a] & = & g'_2\ a \\
G_2\ (x'_1 +\!\!+ x'_2) & = & G_2\ x'_2 \otimes G_2\ x'_1
\end{array}
$$

It is not difficult to check that $g_0, g_1, g'_2, \wedge, \otimes$ can be implemented using constant parallel time. It soon follows from the performance lemma that the final program is an $O(\log n)$ parallel program. To be more explicit, we summarize our final algorithm in Figure 1, by applying the tupling transformation, and substituting all functions with their final definitions.

In fact, as discussed in [13], this program can be automatically translated into an efficient NESL code [3], which can run on various of parallel architectures in practice. The NESL code is given in the appendix.

5. Conclusion

In this paper, we show, by a case study, that the parallelization lemma is very useful for formal derivation of parallel algorithms, besides its effectness in constructing parallelizing compiler as studied in [15]. In particular, we formally derive a novel

but efficient parallel homomorphic algorithm for bracket matching, which would be difficult with other approaches [10,11].

Acknowledgements

Thanks are to Wei-Ngan Chin for stimulating discussions on parallelization, and to Manuel M. T. Chakravarty and Gabriele Keller for helping us to test on a Cray machine the final NESL code in the appendix.

References

[1] D. Barnard, J. Schmeiser, and D. Skillicorn. Deriving associative operators for language recognition. In *Bulletin of* EATCS (43), pages 131–139, 1991.

verlag.

programming with list homomorphisms. *Parallel Processing Letters*,

W. Rytter. *Efficient Parallel Algorithms*. Cambridge University

third homomorphism theorem. *Journal of Functional Programming*,
96.

structing list homomorphisms. Technical Report MIP-9512, Fakultt fr
Informatik, Universitt Passau, August 1995.

ematic efficient parallelization of scan and other list homomorphisms.
pean Conference on Parallel Processing, LNCS 1124, pages 401–
yon, France, August 1996. Springer-Verlag.

and P. Harrison. Parallelism via homomorphism. *Parallel Processing*
–295, 1996.

ki, and M. Takeichi. Formal derivation of efficient parallel programs
of list homomorphisms. *ACM Transactions on Programming Lan-*
tems, 19(3):444–461, 1997.

i, and M. Takeichi. Implementing homomorphisms in parallel. In *15th*
n Soc. for Software Sci. and Tech., pages D4–2, Japan, September

ki, M. Takeichi, and A. Takano. Tupling calculation eliminates mul-
sals. In *ACM SIGPLAN International Conference on Functional*
ages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

chi, and W.N. Chin. Parallelization in calculational forms. In *25th*
m on Principles of Programming Languages, pages 316–328, San
, USA, January 1998.

Fegaras. A fold for all seasons. In *Proc. Conference on Functional*
anguages and Computer Architecture, pages 233–242, Copenhagen,

[6] M. Cole. Parallel
5(2), 1995.

[7] A. Gibbons and
Press, 1988.

[8] J. Gibbons. The
6(4):657–665, 199

[9] S. Gorlatch. Con
Mathematik und

[10] S. Gorlatch. Syst
In *Annual Euro*
408, LIP, ENS L

[11] Z.N. Grant-Duff
Letters, 6(2):279

[12] Z. Hu, H. Iwasa
by construction
guages and Sys

[13] Z. Hu, H. Iwasak
Conf. Proc. Jp
1998.

[14] Z. Hu, H. Iwasa
tiple data travers
Programming, p

[15] Z. Hu, M. Takei
ACM Symposiu
Diego, California

[16] T. Sheard and L.
Programming L
June 1993.

Appendix A

The following gives our NESL code for Bracket Matching. It is an $O(\log n)$ parallel program according to the performance model of NESL.

```
% ======== Stack Datatype ===========%

datatype Stack ([Char], Int);

function isEmpty (s) : Stack - Bool =
  let
    Stack (pu, po) = s
  in
  (#pu == 0 and po == 0);

function noEmpty (s) : Stack - Bool =
  not (isEmpty (s));

function push (c, s) : (Char, Stack) - Stack =
  let
    Stack (pu, po) = s
  in
  Stack ([c] ++ pu ,po);

function pop (s) : Stack - Stack =
  let
    Stack (pu, po) = s
  in
  if #pu == 0
  then
    Stack (pu, po + 1)
  else
    Stack (drop (pu, 1), po);

function top (s) : Stack - Char =
  let
    Stack (pu, po) = s
  in
  pu[0];

empty = Stack (dist ('x, 0), 0);

function otimes (x, y) : (Stack, Stack) - Stack =
  let
    Stack (xpu, xpo) = x;
    Stack (ypu, ypo) = y
  in
  if xpo >= #ypu
  then
    Stack (xpu, xpo + ypo - #ypu)
  else
    Stack (xpu ++ drop (ypu, xpo), ypo);

% ========== Main Algorithm ==========%

function otimes_rev (x, y) : (Stack, Stack) - Stack =
  otimes (y, x);

function isOpen (c) : Char - Bool =
  (c == '( or c == '[ or c == '{);

function isClose (c) : Char - Bool =
  (c == ') or c == '] or c == '});

function match (c1, c2) : (Char, Char) - Bool =
  (c1 == '( and c2 == ')) or
  (c1 == '[ and c2 == ']) or
  (c1 == '{ and c2 == '});

function bm (x, st) : ([Char], Stack) - Bool =
  let
    g20 = {if isOpen (a)
           then
             push (a, empty)
```

```
        else if isClose (a)
        then
          pop (empty)
        else
          empty
      : a in x};
  (g21, e) = scan (otimes_rev, empty, g20);
  s0  = {otimes (g21e, st) : g21e in g21};
  b0  = {if isOpen (a)
        then
          F
        else
          if isClose (a)
          then
            (if noEmpty (s)
             then
               (match (top (s), a) and isEmpty (pop (s)))
             else
               F)
          else
            isEmpty (s)
      : s in s0; a in x};
  g10 = {if isOpen (a)
        then
          T
        else if isClose (a)
        then
          (if noEmpty (s) then match (top (s), a) else F)
        else
          T
      : s in s0; a in x};
  g11 = and_scan (g10);
  b1  = {g11e and b0e : g11e in g11; b0e in b0}
in
b1[#b1 - 1];

function bracketMatching (str) : [Char] - Bool =
  bm (str, empty);
```