On the parallel complexity of loops

Oscar H. Ibarra *, Nicholas Q. Tran, Tao Yang

Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

Received September 1995; revised May 1996
Communicated by A. Salomaa

Abstract

Many methods have been proposed to parallelize loops for different scenarios using various syntax and/or semantics analysis techniques. An interesting fundamental question is whether one can build a "general" compiler which is able to produce an efficient parallel algorithm for every instance of a nested loop. We give a theoretical analysis using the PRAM complexity theory and present some positive and negative results.

1. Introduction

The problem of loop parallelization has been addressed extensively in the literature [11, 16, 21]. There are many loop transformation and partitioning techniques proposed to uncover loop parallelism and take advantage of memory hierarchy and coarse-grain computation (e.g. loop interchange, skewing, tiling, [3, 11, 21]) If a semantics analysis could identify the meaning of a loop, then more optimization can be conducted, for example, parallelizing a loop for global summation [21]. For some applications, run-time compilation [13, 15] could be used to explore parallelism. Considering other related existing or future work dealing with compile and run-time loop parallelization, one might be led to believe that a parallelizing compiler can be developed eventually, which would produce efficient parallelization of all loops. We conduct an analysis on such a possibility using the PRAM theory. We formally show that it is unlikely (in some precise sense, to be defined later) that a "general" compiler can be constructed that is able to produce an efficient parallel code for every instance of a doubly nested loop. However, we show that it is possible for some single loops. It should be noted that "general" here means that the compiler is allowed to use any technique as long as the input/output behavior of the code produced by the compiler is identical to the behavior of the given nested loop. This result may be intuitively true, but as far as we...
know, no formal analysis has appeared in the literature. Our contribution is to provide a formal analysis on the complexity of this loop parallelization problem.

2. The complexity of loops

As we mentioned in Section 1, there are various efforts in developing compilation techniques to parallelize loops for different cases. Restructuring techniques such as skewing, loop interchanging [11,21] can explore more parallelism. These techniques preserve data dependence. There are more aggressive methods which break data dependence but still preserve the input–output semantics of a loop. For example, the privatization technique [13] introduces temporal local variables for each processor to relax data dependence constraint and explore more parallelism. An application example is shown in the left part of Fig. 1. Also, a semantic analysis [21] may be able to identify the special functions of a loop and replace it with an efficient implementation of a parallel algorithm, for example, global summation as shown in the right part of Fig. 1. Recently, run-time compilation techniques [13,15] have been proposed to provide more avenues to parallelize loops which contain unknown information at compile time.

In summary, semantic-preserving compile-time or run-time techniques for exploring loop parallelism have been investigated extensively. We expect that compiling of loops will be still a major research focus since loops contain most parallelism in application programs, and more sophisticated techniques will be developed. An interesting question is whether a general compiler can be developed eventually that incorporates sophisticated techniques that can produce an efficient parallel algorithm for every instance of a nested loop. An answer to this fundamental question will demonstrate the limits that a compiler can achieve. Notice that we are not concerned about the complexity of the compiler, i.e., the compiler or run-time compilation scheme could spend whatever time it needs. We are interested only in the question of whether the result of the compilation (i.e., the parallel algorithm generated) is always efficient. In this section, we give circumstantial evidence that the answer is negative. Specifically, we show that it is unlikely that a “general” compiler can be constructed that is capable of producing an efficient parallel algorithm (with the same input/output semantics) for every instance of a doubly nested loop.

We use the PRAM and complexity theory to conduct a formal analysis. To be precise, define EREW\(i\) to be the class of problems solvable by Exclusive Read Exclusive Write PRAMs in \(O(\log^n)\) time using a polynomial number of processors, and \(NC = \bigcup_{i \geq 1} EREW^i\). CREW\(i\), ERCW\(i\) and CRCW\(i\) are defined similarly. There is an

\[
\text{for } i = 1 \text{ to } n \\
\text{temp} = y[i]; y[i] = x[i]; x[i] = \text{temp} \\
\text{endfor}
\]

\[
\text{for } i = 1 \text{ to } n \\
x = x + y[i] \\
\text{endfor}
\]

Fig. 1. Both programs cannot be parallelized if the data dependence is followed strictly. But the left one can be parallelized by privatization and the right by semantics analysis.
alternative definition of NC using Boolean circuits. If $\text{NC}^i$ is the class of problems solvable by uniform Boolean circuits of polynomial size ($= \text{number of gates}$) and depth $O(\log^i n)$, then $\text{NC} = \bigcup_{i \geq 1} \text{NC}^i$ [2, 14].

It is known that $\text{NC}^i \subseteq \text{EREW}^i \subseteq \text{CREW}^i \subseteq \text{CRCW}^i \subseteq \text{NC}^{i+1}$ [9]. Thus, NC under (any of) the PRAM models is identical to NC under the Boolean circuit model, although the refined classes, e.g., $\text{EREW}^i$ and $\text{NC}^i$, may not be identical. As we are interested in lower bounds, we state the results using the Boolean circuit model of parallel computation.

The situation is similar to that for sequential computation. The class of problems solvable by RAMs (random access machines) in polynomial time is identical to the class of problems solvable by deterministic Turing machines in polynomial time. This class is called $\text{P}$.

Whereas $\text{P}$ can be thought of as the class of problems that are feasible (tractable) under the sequential model of computation, we can think of $\text{NC}$ as the class of problems that can be solved efficiently under the parallel model of computation. In particular, $\text{NC}^1$ is the class of problems that admit the fastest parallel algorithms.

Although the models for the study of NC may not be too practical because they are assumed to have a polynomial (in the size of the input) number of processors, the results have practical implications when dealing with the more practical models, where the number of processors is fixed, independent of the size of the problem, i.e., even when the number of processors is fixed, it is reasonable to expect that, generally, the problems in NC will admit more efficient parallel algorithms than those that are not in NC.

It is easy to show that NC is contained in $\text{P}$. The converse is not known, although it is widely conjectured that $\text{P}$ is not contained in NC. A problem is $\text{P}$-complete if it is in $\text{P}$, and it has the property that if it is in NC, then $\text{P}$ is contained in NC, i.e., $\text{P} = \text{NC}$. Thus, one can think of a $\text{P}$-complete problem as a representative of the hardest (with respect to admitting efficient parallel algorithms) problems in $\text{P}$.

We now make precise the meaning of "unlikely" in the paper.

**Definition.** Though it is still an open problem, it is widely believed that $\text{P}$ is not contained in NC. It is in this sense that we use the term "unlikely". Thus, e.g., when we use the phrase "is unlikely in NC" or "unlikely to admit an efficient parallel algorithm", we are assuming that most likely $\text{P}$ is not contained in NC, as conjectured.

Many $\text{P}$-complete problems have been shown (see, e.g., [18, 10]). Here we show that doubly-nested loops can solve $\text{P}$-complete problems and, hence, there computations are not likely to be efficiently parallelizable.

**Notation.** In the sequel, $A$ is an alphabet (i.e., a finite set of symbols), $R_0$ is a subset of $A$, $2^A$ is the set of all subsets of $A$, $\cup$ is set union, and $\cap$ is set intersection; $g, g_1, g_2, \ldots$ are functions from $A \times 2^A$ to $2^A$. For a symbol $a$ and a set of symbols $Q$, we use $g(a, Q)$ to denote the set $\{g(a, x) \mid x \in Q\}$. For notational convenience, let $a_0 = b_0 = \varepsilon$, and the boundary conditions $R(i, -1) = R(-1, i) = D$, where $\varepsilon$ and $D$ are dummy values.
Theorem 1. There is an alphabet $A$, a subset $R_0$, and functions $g_1$ and $g_2$ from $A \times 2^A$ to $2^A$ such that determining the value of $R(n,n)$ computed by the following code is unlikely to be in NC. Note that the input to the code is $n$ (the size of the problem) and symbols $a_1, \ldots, a_n, b_1, \ldots, b_n$ which come from the alphabet $A$.

\[
R(0,0) = R_0;
\quad \text{for } i = 1 \text{ to } n
\quad \text{for } j = 1 \text{ to } n
\quad R(i,j) = g_1(a_i, R(i-1,j)) \cap g_2(b_j, R(i,j-1))
\quad \text{endfor}
\quad \text{endfor}
\]

The proof of Theorem 1, which consists of showing that a doubly nested loop of the above form can solve a P-complete problem, is given in Section 3 (see Corollary 1). However, for the case of single loops, we can show the following using a technique similar to the proof that regular sets are in NC$^1$ [17].

Theorem 2. Determining the value of $R(n,n)$ computed by the following code is in NC$^1$ for any alphabet $A$, subset $R_0$, and function $g$ from $A \times 2^A$ to $2^A$.

\[
R(0) = R_0;
\quad \text{for } i = 1 \text{ to } n
\quad R(i) = g(a_i, R(i-1))
\quad \text{endfor}
\]

In contrast to Theorem 1, when $\cap$ is replaced by $\cup$, we have the following result whose proof is in Section 4 (see Theorem 5).

Theorem 3. Determining the value of $R(n,n)$ computed by the following code is in NC$^2$ for any alphabet $A$, subset $R_0$, and functions $g_1$ and $g_2$ from $A \times 2^A$ to $2^A$. However, it is unlikely to be in NC$^1$ for some alphabet $A$, subset $R_0$, and functions $g_1$ and $g_2$.

\[
R(0,0) = R_0;
\quad \text{for } i = 1 \text{ to } n
\quad \text{for } j = 1 \text{ to } n
\quad R(i,j) = g_1(a_i, R(i-1,j)) \cup g_2(b_j, R(i,j-1))
\quad \text{endfor}
\quad \text{endfor}
\]

It is interesting to note that even with an EREW PRAM, it is not known whether the code in Theorem 3 is in EREW$^1$. However, if $R(i,j)$ in Theorem 3 satisfies the condition that if $R(i,j)$ is not empty, then at least one of $R(i+1,j)$ and $R(i,j+1)$ is empty for all $i$ and $j$, then one can show that the code is in EREW$^1$; however, for
some alphabet $A$, subset $R_0$, and functions $g_1$ and $g_2$, the code is not likely to be $\text{NC}^1$. The proof is given in Section 5 (see Theorem 6). Note that this seems to give some evidence that EREW$^1$ is a larger class than $\text{NC}^1$.

One can generalize $R(i, j)$ to $R(i, j) = g_1(a_i, c_{i+j}, R(i - 1, j)) \cup g_2(b_j, c_{i+j}, R(i, j - 1))$, and it will still be in $\text{NC}^2$.

**Example.** Consider the string shuffling problem [20], which is defined as follows. Given an alphabet $\Sigma$ and three strings $x, y, z \in \Sigma^*$, where $|x| = |y|$ and $|z| = |x| + |y|$, determine whether $z$ is a shuffle of $x$ and $y$. Let $x = a_1a_2 \ldots a_n$, $y = b_1b_2 \ldots b_n$ and $z = c_1c_2 \ldots c_{2n}$. The string shuffling problem can be solved using a doubly nested loop with $R_0 = t$ and $g_1$ and $g_2$ defined below, where $t$ and $\emptyset$ represent true and false, respectively.

$$g_1(a_i, c_{i+j}, R(i - 1, j)) = \{t\} \text{ if } R(i - 1, j) = \{t\} \text{ and } a_i = c_{i+j} \text{ else } \emptyset$$

$$g_2(b_j, c_{i+j}, R(i, j - 1)) = \{t\} \text{ if } R(i, j - 1) = \{t\} \text{ and } b_j = c_{i+j} \text{ else } \emptyset$$

The string $z$ is a shuffle of strings $x$ and $y$ if and only if $R(n, n) = \{t\}$. Hence, the string shuffling problem is in $\text{NC}^2$.

The code in Theorem 3 or in the preceding example can be generalized to have $t$ nested loops, where $R(i_1, \ldots, i_t)$ now depends on $t$ unions of functions $g_1, \ldots, g_t$, and each $g_s$ depends essentially on only one coordinate of $R$. So, for example when $t = 3$, we have $R(i, j) = g_1(a_i, R(i - 1, j, k)) \cup g_2(b_j, R(i, j - 1, k)) \cup g_3(c_k, R(i, j, k - 1))$. This type of nested loops is also in $\text{NC}^2$.

### 3. Recurrence equations that are P-complete

Nested loops of the form given in the previous section can be rewritten in terms of recurrence equations, and vice versa. The conversion between loops and recurrences is obvious. We find it more convenient to use recurrences, and hence will prove the results using recurrence equations.

Consider the following recurrence equation:

$$R(0, 0) = c$$

$$R(i, j) = f(a_i, R(i - 1, j), b_j, R(i, j - 1))$$

for $0 \leq i \leq n, 0 \leq j \leq m$ such that $i + j \geq 1$,

where $c, a_r, \text{ and } b_s (1 \leq r \leq n, 1 \leq s \leq m)$ are from a finite set of constants independent of $n$ and $m$, and the function $f$ depends only on the values of $a_i, R(i - 1, j), b_j, R(i, j - 1)$ and not on the indices $i$ and $j$. We assume without loss of generality that $m \leq n$. For notational convenience, let $a_0 = b_0 = c$, and the boundary conditions $R(0, -1) = R(-1, 0) = d$, where $d$ is a dummy constant. The objective is to compute $R(n, m)$. Note
that in Eq. (1) we can have \( f \) depend also on \( R(i-1,j-1) \); however, this dependence can be removed by a simple coding technique.

Clearly, Eq. (1) can be solved by a parallel algorithm in linear time using a linear number of processors by computing along the diagonals of the recurrence table \( R(i,j) \). We will show that it is unlikely that (1) can be solved by a parallel algorithm in polylogarithmic time, i.e., it is unlikely that it belongs to the class NC.

We will show that there is a recurrence of type (1) that solves a P-complete problem (with respect to log-space reductions). Hence, if such a recurrence is in NC, then \( P \) (= problems solvable by sequential polynomial time algorithms) equals NC, which is widely believed to be unlikely. The proof involves a reduction to a problem concerning reset deterministic linear-bounded automaton [5].

A DLBA is equivalent to a deterministic linear space Turing machine [4]. A reset deterministic linear space Turing machine is a restricted DLBA which operates as follows: the machine starts on the left end of the input tape at \( \ldots a_n \) in a distinguished reset state, \( r \). Then it scans the tape from left-to-right (advancing one tape square to the right in each step), changing states and reading/rewriting the tape just like a DLBA. The machine either halts in an accepting state after processing \( a_n \), or it resets to the left end of the tape in the reset state, \( r \), to begin a new left-to-right sweep.

It is important to note that the reset state is always \( r \). If this were not the case, i.e., if we allow the machine to reset to a different state (e.g., allowing the machine to reset to the state it enters after processing \( a_n \)), then one can easily show that such a machine is equivalent to a DLBA. It is an open problem whether resetting DLBAs are equivalent to DLBAs [5].

A resetting DLBA that makes \( S(n) \) sweeps on the input before accepting is called an \( S(n) \)-sweep resetting DLBA. The following lemma can be shown.

**Lemma 1.** There is an \( n \)-sweep resetting DLBA that accepts a P-complete language.

**Theorem 4.** There is a recurrence equation of type (1) that accepts a P-complete language. Thus, it is unlikely that such a recurrence is in NC.

**Proof.** By Lemma 1, it suffices to show that the computation of an \( n \)-sweep resetting DLBA can be reduced to solving a recurrence of type (1).

Let \( M \) be an \( n \)-sweep resetting DLBA with state set \( Q = \{1,2,\ldots,s\} \), input alphabet \( \Sigma \), worktape alphabet \( \Gamma \), and transition function \( \delta \). Note that \( \Sigma \subseteq \Gamma \), and we assume \( r \in \Gamma \). Assume that the resetting state is \( 1 \) and \( s \) is the only accepting state. Since \( M \) moves right after each atomic move, we can write the transition function in the form \( \delta(q,a) = [q',a'] \). This means that \( M \) in state \( q \) reading \( a \) enters state \( q' \) after rewriting \( a \) by \( a' \).

Given an input \( a_1 \ldots a_n \), we define recurrence \( R \) in the following manner: informally, \( R(i,j) \) represents the pair of state and symbol of \( M \) after processing the \( i \)th tape cell in sweep \( j \). We denote by \( st(R(i,j)) \) the first component of \( R(i,j) \), and \( sym(R(i,j)) \) the
second component. The sweeps are numbered 0, . . . , n − 1 (we assume the 0th position of the tape always contains the symbol \( a_0 = \varepsilon \)).

\[
\begin{align*}
R(0,0) &= [1, \varepsilon] \\
R(i,0) &= f(a_i,R(i-1,0),R(i,-1)) = \delta(st(R(i-1,0)),a_i) \quad \text{for } 1 \leq i \leq n \\
R(0,j) &= f(a_0,R(-1,j),R(0,j-1)) = [1, \varepsilon] \quad \text{for } 1 \leq j \leq n - 1 \\
R(i,j) &= f(a_i,R(i-1,j),R(i,j-1)) = \delta(st(R(i-1,j)),\text{sym}(R(i,j-1))) \\
&\quad \text{for } 1 \leq i \leq n, 1 \leq j \leq n - 1
\end{align*}
\]

Clearly, the recurrence above is of type (1), where the \( b_j \)'s are set to \( \varepsilon \).

**Remark 1.** We can give a converse of the construction above, provided the set of all possible values of \( R(i,j) \) is finite, independent of \( n \) and \( m \). We can show that given a recurrence equation of type (1), we can construct a resetting DLBA to simulate the evaluation of \( R(n,m) \). We sketch the construction of the resetting DLBA \( M \). The input to \( M \) is the string \( b_1 b_2 \ldots b_n a_1 a_2 \ldots a_n \) (of length \( n + m + 1 \)). \( M \) creates a new track on the tape to record \( R(i,j) \) in cell \( i + m + 1 \) in sweep \( j \). For \( M \) to compute \( R(i,j) \), it needs \( a_i, R(i-1,j), b_j, R(i,j-1) \). Now \( a_i \) and \( R(i,j-1) \) are in cell \( i + m + 1 \). \( R(i-1,j) \) was just computed and written in the previous cell, and \( M \) can remember this information in the state. Thus, if \( b_j \) is also available, \( M \) can compute \( R(i,j) \), record it in cell \( i + m + 1 \), and remembers it in the state (replacing \( R(i-1,j) \)). To access \( b_j \), \( M \) can use a marker to mark a tape cell position. The marker is initially placed in position 1 at the beginning of the first sweep (i.e., sweep 0), and \( M \) remembers \( b_1 \) in the state. In the second sweep (sweep 1), \( M \) moves the marker to position 2, thus marking \( b_2 \), and now remembers \( b_2 \) in the state, etc. Thus, \( M \) can compute \( R(n,m) \).

In recurrence equations (1) and (2), \( R(i,j) \) depends on both \( R(i-1,j) \) and \( R(i,j-1) \), i.e., in general, in the function \( f \), \( R(i-1,j) \) and \( R(i,j-1) \) interact. We now look at the special case when the equation is of the form

\[
\begin{align*}
R(0,0) &= R_0, \\
R(i,j) &= g_1(a_i,R(i-1,j)) \op g_2(b_j,R(i,j-1)).
\end{align*}
\]

where

1. \( R_0 \) is a finite set.
2. \( R(i,j) \) is a finite set for all \( i \) and \( j \), and the set of all such sets is finite, independent of \( n \) and \( m \).
3. \( \op \) is an operation on sets.

We will show that even for this case, there is such a recurrence equation that accepts a \( P \)-complete language. We give an example where \( \op \) is set intersection. We can modify the proof of Theorem 1 as follows.
Given an input \( a_1 \ldots a_n \), we define recurrence \( R \) in the following manner: informally, \( R(i,j) \) represents the quadruple of states and symbols of \( M \) before and after processing the \( i \)th tape cell in sweep \( j \). We denote by \( st1(R(i,j)), sym1(R(i,j)), st2(R(i,j)), \) and \( sym2(R(i,j)) \) the first, second, third, and fourth component of \( R(i,j) \), respectively.

\[
R(0,0) = \{[1,1,1,1]\},
\]

\[
R(i,0) = g_1(a_i, R(i - 1, 0)) \cap g_2(\varepsilon, R(i, -1))
= \{[1, a_i, \delta(st2(R(i - 1, 0)), a_i)]\} \cap Q \times \Sigma \times Q \times \Sigma \quad \text{for} \quad 1 \leq i \leq n,
\]

\[
R(0,j) = g_1(a_0, R(-1,j)) \cap g_2(\varepsilon, R(0, j - 1))
= \{[1, \varepsilon, 1, \varepsilon]\} \cap Q \times \Gamma \times Q \times \Gamma \quad \text{for} \quad 1 \leq j \leq n - 1,
\]

\[
R(i,j) = g_1(a_i, R(i - 1,j)) \cap g_2(\varepsilon, R(i, j - 1))
= \{[st2(R(i - 1,j)), \gamma, \delta(st2(R(i - 1,j)), \gamma)]| \gamma \in \Gamma\}
\cap \{[q, sym2(R(i,j - 1)), \delta(q, sym2(R(i,j - 1))))| q \in Q\}
\quad \text{for} \quad 1 \leq i \leq n, 1 \leq j \leq n - 1
\]

Clearly, the recurrence above is of type (3), where the \( b_j \)'s are set to \( \varepsilon \).
Note that because the sweeping machine is deterministic, the result of any intersection is a singleton set. Thus, we have the following corollary, which is equivalent to Theorem 1.

**Corollary 1.** There is a recurrence equation of type (3) that accepts a P-complete language. Thus, it is unlikely that such an equation is in NC.

**4. Recurrence equations that are in NC\(^2\)**

There are recurrence equations of type (1) that are solvable in polylogarithmic time using a polynomial number of processor. For example, recurrences that can be recast as a shortest path problem are in NC\(^2\). For instance, finding the string edit distance of two strings, given the cost functions for \( \text{change, delete, insert} \), can be reduced to solving a recurrence equation of type (1), and the recurrence can be reduced to solving a shortest path problem. For these kinds of recurrences, there is an \( O(\log^3 n) \) time parallel algorithm (see, e.g., [7, 12]). The longest common subsequence problem, the minimum-length time-warping of two sequences, and other string processing problems can also be solved this way.

Another class of recurrences that can be solved in \( O(\log^2 n) \) time has the form of recurrence (3), but now \( \text{op} \) is set union:

\[
R(0,0) = R_0,
\]

\[
R(i,j) = g_1(a_i, R(i - 1,j)) \cup g_2(b_j, R(i, j - 1)).
\]
In contrast to Corollary 1, we have

**Theorem 5.** Recurrence (5) can be solved by a nondeterministic Turing machine using logarithmic space, i.e., it is in NLOGSPACE (= the class of languages accepted by nondeterministic Turing machines using logarithmic space.) Hence, this type of recurrence is in NC² (since NLOGSPACE ⊆ NC² [1, 14]).

**Proof.** We will show the following recognition problem is in NLOGSPACE: “Given \( a_1 \ldots a_n \# b_1 \ldots b_m \# r_1 \ldots r_k \), is \( R(n,m) = \{ r_1, \ldots, r_k \} \) ?”

We first construct a nondeterministic Turing machine \( M_1 \) to recognize the following problem: “Given \( a_1 \ldots a_n \# b_1 \ldots b_m \# r \), is \( r \in R(n,m) \) ?”

On an input of the form above, \( M_1 \) simply guesses a string \( d_1, \ldots, d_{n+m} \), where each \( d_i \) is 1 or 2. \( M_1 \) uses this string to verify that \( r \) is a member of the set

\[
g_{d_{n+m}}(g_{d_{n+m-1}}, \ldots (g_{d_2}(g_{d_1}(R_0,s_1),s_2), \ldots, s_{n+m-1}),s_{n+m}),
\]

where \( s_j = a_j \) if \( d_j = 1 \) and \( s_j = b_j \) if \( d_j = 2 \) for \( 1 \leq j \leq m+n \). Note that \( r \in R(n,m) \) if and only if the above relation is satisfied. Clearly \( M_1 \) needs only \( \log n \) space to record the last bit positions of \( a_1 \ldots a_n \) and \( b_1 \ldots b_m \) that have been processed so far.

From machine \( M_1 \), we can obtain a \( \log n \)-space machine \( M_2 \) to accept the pseudocomplement language, i.e., “Given \( a_1 \ldots a_n \# b_1 \ldots b_m \# r \), is \( r \notin R(n,m) \)” This is possible, since NLOGSPACE is closed under complementation [8].

Using \( M_1 \) and \( M_2 \), we can recognize the original language using the following algorithm:

for every \( r_i \) do
   run \( M_1 \) on \( a_1 \ldots a_n \# b_1 \ldots b_m \# r_i \)
for every \( t \in \text{complement of} \{ r_1, \ldots, r_k \} \) do
   run \( M_2 \) on \( a_1 \ldots a_n \# b_1 \ldots b_m \# t \)
accept if every computation succeeds

**Remark 2.** Later, in Section 5 (Corollary 2) we show that it is unlikely that the NLOGSPACE in Theorem 5 can be replaced by DLOGSPACE = the class of languages accepted by deterministic Turing machines using logarithmic space.

The following generalization of (5) is also in NLOGSPACE and hence in NC².

\[
R(0,0) = R_0,
\]

\[
R(i,j) = g_1(a_i,c_{i+j},R(i-1,j)) \cup g_2(b_j,c_{i+j},R(i,j-1)),
\]

where now we have three sequences \( a_1, \ldots, a_n, b_1, \ldots, b_m \), and \( c_1, \ldots, c_{n+m} \).

Recurrences of the forms (5) and (6) are commonly used in dynamic programming solutions to many problems in pattern matching, sequence comparison, and language recognition. An example is the string shuffling problem discussed at the end of Section 2.
Eqs. (5) and (6) can be generalized to $t$ dimensions, where $R(i_1, \ldots, i_t)$ now depends on $t$ functions $g_1, g_2, \ldots, g_t$, and each of these functions depends only on one coordinate of $R$. This type of recurrence equations is also in NLOGSPACE, and hence in NC$^2$.

5. Recurrence equations not likely to be in NC$^1$

We will now show that it is unlikely that (5) is in NC$^1$. Specifically, we will show that (5) is in NC$^1$ if and only if DLOGSPACE = NC$^1$, and the latter is a well-known open problem. This result holds, even if we restrict recurrence (5) so that for all $i$ and $j$, if $R(i,j)$ is not empty, then at least one of $R(i+1,j)$ and $R(i,j+1)$ is empty. We call this restricted type (5').

We will show that (5') is NC$^1$ if and only if DLOGSPACE = NC$^1$. Suppose DLOGSPACE = NC$^1$. Since any recurrence of type (5') is obviously solvable by a log-space deterministic Turing machine due to the restriction on exactly one of $R(i-1,j)$ and $R(i,j-1)$ being nonempty, it follows that (5') is in NC$^1$. To prove the converse, we first reduce the problem to the membership question for one-way two-tape deterministic finite automaton (1dfa(2-tapes)).

A 1dfa(2-tapes) $M$ is a dfa with two one-way read-only input tapes (1 head/tape). We assume that exactly one head moves to the right on each atomic move, and the state dictates which head reads an input symbol, i.e., there is a head selector function $h : Q \rightarrow \{1,2\}$, where $h(q) = i$ means head $i$ does the reading. $M$ accepts a pair of strings $(a_1 \ldots a_n, b_1 \ldots b_m)$ from some fixed alphabet if $M$ when given the input $(a_1 \ldots a_n, b_1 \ldots b_m)$ starting in a distinguished start state with the heads to the left of $a_1$ and $b_1$, eventually enters an accepting state with both heads to the right of $a_1 \ldots a_n$ and $b_1 \ldots b_m$, respectively. Let the states be $1, 2, \ldots, s$, where $1$ is the start state, and $s$ is the only accepting state.

The language accepted by $M$ consists of all pairs $(x, y)$ accepted by $M$ and is denoted by $L(M)$. The membership question is: given a pair $(x, y)$, is it in $L(M)$?

Lemma 2. Deciding if $(a_1 \ldots a_n, b_1 \ldots b_m)$ is accepted by $M$ can be reduced to solving a recurrence equation of the form (5').

Proof. Let $(q, i, j)$ denote the configuration wherein $M$ is in state $q$, and heads 1 and 2 have just processed symbols $a_i$ and $b_j$ (i.e., the heads are on positions $i+1$ and $j+1$ on their respective tapes). Now define $\bar{R}(i, j) = q$ if and only if $M$ can enter configuration $(q, i, j)$ from the initial configuration $(1, 0, 0)$. Thus, $M$ accepts $(a_1 \ldots a_n, b_1 \ldots b_m)$ if and only if $R(n, m)$ is the accepting state, $s$. Since exactly one head moves to the right on each atomic move, $R(i, j)$ can be evaluated according to the following recurrence:

$$ R(0, 0) = 1,$$

$$ R(i, j) = g_1(a_i, R(i-1, j)) \cup g_2(b_j, R(i, j-1)) $$

for $0 \leq i \leq n, 0 \leq j \leq m$ such that $i + j \geq 1$, (7)
where
\[
g_1(a_i, R(i-1, j)) = \begin{cases} 
  p & \text{if } R(i-1, j) = q, h(q) = 1, \text{ and } \delta(q, a_i) = p, \\
  \emptyset & \text{if } R(i-1, j) \text{ is undefined,} 
\end{cases}
\]
\[
g_2(b_j, R(i, j-1)) = \begin{cases} 
  p & \text{if } R(i, j-1) = q, h(q) = 2, \text{ and } \delta(q, b_j) = p, \\
  \emptyset & \text{if } R(i, j-1) \text{ is undefined.} 
\end{cases}
\]

Note that since the machine $M$ is deterministic, exactly one of $R(i-1, j)$ and $R(i, j-1)$ is defined and is a singleton for all $i$ and $j$. Also, since the number of states is finite, the set of all possible values of $R(i, j)$ is finite, independent of $n$ and $m$. Hence, (7) is of the form (5r). \qed

Thus, it is sufficient to prove that if the membership question for $1\text{dfa}(2\text{-tapes})$ is in $\text{NC}^1$, then $\text{DLOGSPACE} = \text{NC}^1$.

We can further reduce the problem to showing that if the membership question for one-way two-head deterministic finite automaton ($1\text{dfa}(2\text{-heads})$) is in $\text{NC}^1$, then $\text{DLOGSPACE} = \text{NC}^1$. This is because, the computation of a $1\text{dfa}(2\text{-tapes})$ when given input $(x, x)$, i.e., the two tapes are identical, is essentially the computation of a $1\text{dfa}(2\text{-heads})$ operating on input $x$. The technique of reducing a multi-head computation to a two-head computation in the following lemma has been used before in [16a].

**Lemma 3.** The membership question for $1\text{dfa}(2\text{-heads})$ (i.e., for a fixed machine $M$ and an arbitrary input $x$ of length $n$, is $x$ in the language accepted by $M$?) is in $\text{NC}^1$ if and only if $\text{DLOGSPACE} = \text{NC}^1$.

**Proof.** Suppose $\text{DLOGSPACE} = \text{NC}^1$. Since every language accepted by a $1\text{dfa}(2\text{-heads})$ is in $\text{DLOGSPACE}$, it is also in $\text{NC}^1$. Conversely, suppose every language accepted by a $1\text{dfa}(2\text{-heads})$ is contained in $\text{NC}^1$. We show that if $L$ is accepted by a two-way $k$-head deterministic finite automaton ($2\text{dfa}(k\text{-heads})$) for some $k \geq 2$, then $L$ is in $\text{NC}^1$. Since $\text{DLOGSPACE} = \bigcup_{k \geq 1} \text{languages accepted by } 2\text{dfa}(k\text{-heads})$, this proves $\text{DLOGSPACE}$ is contained in $\text{NC}^1$.

We prove the above claim for the case $k = 3$. The proof directly generalizes to larger values of $k$.

Suppose $L$ is accepted by a $2\text{dfa}(3\text{-heads})$ $M$. For each $x = b_1 b_2 \ldots b_n \in L$, define

\[
x_1 = (x\#)^n,
\]
\[
x_2 = (b_1^n \# b_2^n \# \ldots b_n^n \#)^n,
\]
\[
x_3 = (b_1^n \#)^n (b_2^n \#)^n \ldots (b_n^n \#)^n.
\]

The words $x_1, x_2,$ and $x_3$ are of the same length. Next, define $x_0$ to be the three-track word composed from $x_1, x_2,$ and $x_3$. We call $x_0$ a block. Finally, define

\[
x' = (x_0 \$ 0^n 1^n 0^n 1^n 0^n \ldots -n^n \$)^n.
\]
For example, if \( x = ab \), then
\[
\begin{align*}
x_1 &= ab\#ab\#ab\#ab\#, \\
x_2 &= aa\#bb\#aa\#bb\#, \\
x_3 &= aa\#aa\#bb\#bb\#,
\end{align*}
\]
\[
ab\#ab\#ab\#ab\#\l_0011_1001_1110000\l (repeated 8 times)
\]
\[
x' = aa\#bb\#aa\#bb\#\l_0011_1001_1110000\l (repeated 8 times)
\]
\[
aa\#aa\#bb\#bb\#\l_0011_1001_1110000\l (repeated 8 times)
\]

The blocks of symbols between the dollar signs are called counting blocks.

Next we define a ldfa(2-heads) \( N \) such that on well-formed inputs (inputs of the form \( x' \) for some \( x \)), \( N \) simulates the computation of \( M \) on \( x \). \( N \) will use one head to code the positions of the 3 heads of \( M \). We call this the main head of \( N \). The other head of \( N \) is used only for counting. Thus \( N \) accepts, among other strings, the "padded" ("translated") version of \( L \). Translational techniques have been used in various places in the literature.

At the beginning both heads of \( N \) start at the leftmost symbol of the first block of \( x' \).

Suppose \( M \) moves head \#1 to the right (left). Then \( N \) moves its main head \( n_3 + n_2 + 1(-1) \) steps to the right (not counting the steps when the main head is on the counting blocks), using the other head and the counting blocks to count the steps.

Suppose \( M \) moves head \#2 to the right (left). Then \( N \) moves its main head \( n_3 + n_2 + n(-n) \) steps to the right.

Suppose \( M \) moves head \#3 to the right (left). Then \( N \) moves its main head \( n_3 + n_2 + n^2(-n^2) \) steps to the right.

It is easy to see that the main head of \( N \) reads the same input symbols as the heads of \( M \). Since \( M \) makes at most \( n_3 \) moves, there are enough blocks in \( x' \) for the simulation.

Let \( L' \) be the language accepted by \( N \). Note that \( N \) may accept or reject inputs that are not well-formed. We only need the fact that \( M \) accepts a string \( x \) iff \( N \) accepts \( x' \).

By assumption, every language accepted by a ldfa(2-heads) is in \( NC^1 \), and so there is a uniform family of boolean circuits \( C_1 \) of depth \( O(\log n) \) that recognizes \( L' \). It is easy to construct another uniform family of boolean circuits \( C_2 \) of depth \( O(\log n) \) that will map any \( x \) to \( x' \). By composing \( C_2 \) to \( C_1 \), we get a uniform family of boolean circuits \( C \) of depth \( O(\log n) \) that accepts \( L \).

Hence, if every language accepted by a ldfa(2-heads) is in \( NC^1 \), then every language accepted by a 2dfa(3-heads) is in \( NC^1 \). Since the above argument can be extended to work for any \( k \geq 2 \), this shows that DLOGSPACE \( \subseteq NC^1. \) \( \square \)
Thus, from Lemmas 2 and 3 and the discussion at the beginning of this section, we have

**Theorem 6.** There is a recurrence equation of type (5r) that accepts a language $L$ with the property that $L$ is in $\text{NC}^1$ if and only if $\text{DLOGSPACE} = \text{NC}^1$. Thus, it is unlikely that such an equation is in $\text{NC}^1$.

**Corollary 2.** Recurrences of type (5) are in $\text{DLOGSPACE}$ if and only if $\text{DLOGSPACE} = \text{NLOGSPACE}$.

**Proof.** If $\text{DLOGSPACE} = \text{NLOGSPACE}$, then recurrence (5) is in $\text{DLOGSPACE}$, by Theorem 5. Conversely, suppose recurrence (5) is in $\text{DLOGSPACE}$. It is known that $\text{NLOGSPACE} = \bigcup_{k \geq 1} \text{languages accepted by 2nfa(k-heads)}$, where $n$ denotes nondeterministic. Suppose $L$ is a language accepted by a 2nfa(k-heads). Using a construction similar to the one described in Lemma 3, we can construct an $L'$ accepted by a 1nfa(2-heads), where $L'$ is a padded version of $L$. Since the computation of a 1nfa(2-heads) can be expressed in terms of recurrence (5), and the transformation from $L$ to $L'$ can be done by a log$n$-space deterministic Turing machine transducer, $L$ is in $\text{DLOGSPACE}$. □

There are other types of recurrence equations for which similar results can be shown using automata-theoretic constructions. For example, consider the following recurrence:

$$R(0,0) = R_0, \quad (8)$$

$$R(i,j) = g_1(a_i, R(i-1, j-1)) \cup g_2(a_i, R(i-1, j)) \cup g_3(a_i, R(i-1, j+1)),$$

where $1 \leq i \leq n$, and $0 \leq j \leq \lfloor n/2 \rfloor$. We assume $R(-1,j) = \emptyset$ for all $j$, and $g_k(a_i, \emptyset) = \emptyset$ for $k = 1, 2, 3$. The objective is to compute $R(n,0)$.

Clearly, as in the proof of Theorem 5, we can show that the above recurrence is in $\text{NC}^2$. To show that recurrence (8) is not likely to be in $\text{NC}^1$, even for the case when exactly one of $R(i-1, j-1)$, $R(i-1, j)$, and $R(i-1, j+1)$ is not empty for all $i$ and $j$, we reduce this problem to the membership question concerning deterministic counter machines (DCMs).

A DCM is a one-way dfa augmented with a counter, which is initially set to 0. We assume that on each atomic move the input head moves to the right, and that acceptance occurs when the machine has processed all the input symbols and the counter is 0. (The counter, which always contains a nonnegative integer, can be incremented by 0, 1, -1, and can be tested for 0.)

A configuration of $M$ on input $a_1a_2\ldots a_n$ is a 3-tuple $(q,i,j)$, where $q$ is in $Q$ (= state set of $M$), the machine has just processed input symbol $a_i$, and the counter value is $j$. Assume $Q = \{1,2,\ldots,s\}$, with start state 1 and accepting state $s$. The initial and accepting configurations are $(1,0,0)$ and $(s,n,0)$ respectively. Note that since the input head moves right on each atomic move, if configuration $(r,t,f)$ leads to the accepting configuration, then $j \leq i$ if $i \leq \lfloor n/2 \rfloor$, else $j \leq n - i$ if $i > \lfloor n/2 \rfloor$. Since the machine is
deterministic, a configuration \((r, i, j)\) can be derived from exactly one of three possible predecessors: \((r', i - 1, j - 1)\), \((r', i - 1, j)\), or \((r', i - 1, j + 1)\), i.e., the path from \(R(0, 0)\) to \(R(n, 0)\) is unique. Clearly, one can express the computation of \(M\) in terms of a recurrence of type (8).

It remains to show that the computation of a \(1\text{dfa}(2\text{-heads})\) can be simulated by a DCM. More precisely, let \(L\) be accepted by a \(1\text{dfa}(2\text{-heads})\) \(M\). We construct a DCM \(M'\), which when given a string of the form

\[
x' = \bar{a}_1a_1a_2a_1a_2\ldots a_n\bar{a}_2a_1a_2\ldots a_n\bar{a}_3\ldots \bar{a}_na_1a_2\ldots a_n\$
\]

accepts \(x'\) if and only if \(x = a_1a_2\ldots a_n\) is accepted by \(M\). There are \(n\) blocks in the string \(x'\). If the input heads of \(M\) are on positions \(i\) and \(j\) respectively, then the input head of \(M'\) is on the \((i + 1)\)th symbol of block \(j\). To simulate a move of input head 1 of \(M\), \(M'\) simply moves its input head one step to the right. To simulate a move of input head 2 of \(M\), \(M'\) moves its input head \(2n + 1\) steps to the right, reading off the barred symbol of the next block during the process. The counter and the dollar signs are used to find the correct position of the input head of \(M'\) within the new block.

Clearly, the reduction of \(x\) to \(x'\) can be done in \(\text{NC}^1\). It follows from Lemma 3 that a recurrence equation of type (8) is in \(\text{NC}^1\) if and only if \(\text{DLOGSPACE} = \text{NC}^1\), which is unlikely.

**Remark 3.** It is interesting to note that if the counter of the DCM makes only a fixed number of reversals, i.e., alternations between increasing and decreasing modes, then the language the machine accepts is in \(\text{NC}^1\) [6]. This corresponds to a recurrence of type (8) in which the unique path from \(R(0, 0)\) to \(R(n, 0)\) has only a fixed number of "bends".

### 6. Concluding remarks

We have examined the general question of whether efficient loop parallelization is always possible by using *any* compilation technique. We have provided a formal analysis to show how the PRAM theory can be applied to study the positive and negative aspects of the possibility. While our results are more of a theoretical interest, they demonstrate the limit that a compiler or a run-time compilation scheme can achieve and indicate that it is unlikely that we can find a general technique that will work for all loops. What we can hope for is the development of techniques that will handle large classes of loops.

### Acknowledgements

This work was supported in part by NSF Grant CCR89-18409 (Ibarra) and NSF RIA CCR-9409695 (Yang).
References