A THEORY OF STRICT P-COMPLETENESS

Anne Condon

Abstract. A serious limitation of the theory of P-completeness is that it fails to distinguish between those P-complete problems that do have polynomial speedup on parallel machines from those that don't. We introduce the notion of strict P-completeness and develop tools to prove precise limits on the possible speedups obtainable for a number of P-complete problems.
Key words. Parallel computation; P-completeness.
Subject classifications. 68Q15, 68Q22.

1. Introduction

A major goal of the theory of parallel computation is to understand how much speedup is obtainable in solving a problem on parallel machines over sequential machines. The theory of P-completeness has successfully classified many problems as unlikely to have polylog time algorithms on a parallel machine with a polynomial number of processors. However, the theory fails to distinguish between those P-complete problems that do have significant, polynomial speedup on parallel machines from those that don't. Yet this distinction is extremely important from a practical point of view (see Snyder 1986). In this paper, we refine the theory of P-completeness to obtain *strict* completeness results for a number of P-complete problems.

Kruskal *et al.* (1990) identified the *speedup* of a problem as a critical parameter of the problem. This is simply the ratio between its sequential running time and its parallel running time on a machine with a polynomial number of processors. A problem has *polynomial speedup* if its speedup is $\Omega(n^\epsilon)$, for some $\epsilon > 0$. Given the practical importance of polynomial speedups, Kruskal *et al.* (1990) introduced a new complexity class, SP ("semi-efficient, polynomial

time" or "semi-efficient, parallel") of problems which have polynomial speedup. In earlier work, Simons and Vitter (1986) introduced similar complexity classes and showed that versions of the Circuit Value Problem, Ordered Depth First Search and Unification, where the underlying graphs are "dense", have parallel algorithms with running time approximately the square root of the input size. Another example of a problem in SP is that of computing the expected cost of a finite horizon Markov decision process, a well studied problem in optimization theory (we discuss this problem in detail in Section 5).

However, other P-complete problems seem not to have polynomial speedups on PRAMS with a polynomial number of processors. The general Circuit Value Problem, for example, has no known parallel algorithm on a PRAM with a polynomial number of processors that runs in time $O(n^{1-\epsilon})$ for any $\epsilon > 0$. Proving any polynomial lower bound on the parallel complexity of this problem would separate NC from P. Instead, we ask whether the theory of P-completeness can be refined to prove that the Circuit Value Problem does not have polynomial speedup unless *all* problems in P have polynomial speedup.

To address this question, we introduce the notion of *strict P-completeness*. Roughly, a problem is strictly $T(n)$-complete for P if there is a parallel algorithm for the problem with running time that is within a polylog factor of $T(n)$ and moreover, the existence of a parallel algorithm that improves this by a polynomial factor would imply that *all* problems in P (with at least linear running time) have polynomial speedup. We use this notion to investigate limits on the speedups of P-complete problems.

Our first strictly P-complete problem is a restriction of the Circuit Value Problem (CVP)—the *Square* Circuit Value Problem. A square circuit is a synchronous circuit in which the number of gates at every level is equal to the depth of the circuit. We prove that this problem is strictly $\sqrt{n}$-complete for P.

We apply this result to obtain a similar result for the nonstationary, finite horizon, Markov decision process problem. This problem is to decide the expected cost of a nonstationary Markov decision process with m states in a given time T. Dynamic programming is a sequential algorithm for this problem but the parallel version requires $\Omega(T)$ time. We show that, unless all problems in P have polynomial speedup, any parallel algorithm for the nonstationary Markov decision process problem requires $\Omega(T^{1-\epsilon})$ time on a PRAM with a polynomial number of processors, for all $\epsilon > 0$. We also describe limits on the parallel speedup obtainable for a number of other P-complete problems, including First Fit Bin Packing, Lex First Maximal Independent Set, Ordered Depth First Search and Unification.

We use the RAM and PRAM as our models of sequential and parallel com-

putation, respectively, with a log cost for instructions. The log cost RAM is a model of sequential computation whose cost reflects accurately the actual cost of solving a problem on a real machine (see Cook and Reckhow 1973). Although the PRAM model has been criticized as being a poor practical model of computation (see, for example, Snyder 1986), it is nevertheless an ideal model for proving completeness results, since if polynomial speedup is not obtainable for a problem on a PRAM, it is also not obtainable on more realistic models of parallel computation. There are many variations of the PRAM model, depending on the way simultaneous access to global memory is resolved. Our results are robust in that they hold for all the standard variations of the PRAM model, such as Exclusive Read, Exclusive Write or Priority Concurrent Read, Concurrent Write (see Karp and Ramachandran 1988 for definitions of these models).

Our completeness proof for the Square Circuit Value Problem maps a RAM running in time $t(n)$ to a synchronous circuit of width and depth that are within a polylog factor of $t(n)$. Thus, the size of our circuit is significantly smaller than that obtainable from previous results. By combining the reduction of Ladner (1975) from a Turing machine computation to the Circuit Value Problem and simulations of RAMs by Turing machines of Cook and Reckhow (1973), a RAM running in time $t(n)$ can be mapped to a circuit with depth and width $\Omega(t^2(n))$. Obtaining a synchronous circuit then further squares the size of the circuit (see Greenlaw *et al.* 1994).

The rest of the paper is organized as follows. In Section 2, we give some background on sequential and parallel models of computation. In Section 3, we define precisely our notion of strict P-completeness. In Section 4, we show how to simulate a RAM by a square circuit whose depth is at most a polylog factor times the running time of the RAM, and prove that the Square Circuit Value Problem is strictly $\sqrt{n}$-complete. In Section 5, we extend the results on the Square Circuit Value Problem to obtain limits on the parallel speedups of a number of other P-complete problems.

2. Definitions and background

In this section, we define our models of sequential and parallel computation. We introduce a restricted RAM model in Section 2.1, and describe properties of this model that are needed to prove the main results of the paper. The PRAM model of parallel computation and parallel complexity classes are discussed in Section 2.2. In Section 3, we define the notion of strict P-completeness.

2.1. RAMS and restricted RAMS. Our RAM model is essentially the log cost random access machine of Cook and Reckhow (1973). This model charges for instructions based on the number of bit operations. A RAM consists of a program, or finite sequence of instructions, which accesses an unbounded sized memory of cells $R_0, R_1, \ldots$ Each cell can hold an unbounded length integer. The instruction set we use is given in the following table, along with instruction costs. Logarithms are in base 2.

Instruction	**Cost**
1. $R_i \leftarrow C$	1
2. $R_i \leftarrow R_j + R_k$	$\log R_j + \log R_k$
3. $R_i \leftarrow R_j - R_k$	$\log R_j + \log R_k$
4. $R_i \leftarrow R_{R_j}$	$\log R_j + \log R_{R_j}$
5. $R_{R_j} \leftarrow R_i$	$\log R_j + \log R_i$
6. **goto** L	1
7. **halt**	1
8. **if** $R_i > 0$ **then** *instruction*	$\log R_i$ (+ cost(*instruction*) if $R_i > 0$)
9. **if** $R_i = 0$ **then** *instruction*	$\log R_i$ (+ cost(*instruction*) if $R_i = 0$)

We assume that the input is stored in contiguous memory cells, one bit per cell. Initially the input length is in R_0. We view RAMs as language acceptors and assume that the output bit is stored in R_0. The running time of a RAM on a fixed input is the total cost of the instructions executed on that input. We say a problem, or language, has *sequential running time* $t(n)$ if there is a RAM that, for all inputs of length n, has running time at most $t(n)$. We say that a RAM M *simulates* RAM M' if both M and M' accept the same language.

Cook and Reckhow (1973) showed that the largest value computed by M, and hence the largest address referenced by M, is $2^{O(\sqrt{t(n)})}$. Thus, the cells accessed by a RAM running in time $t(n)$ could have addresses of length $\Omega(\sqrt{(t(n))})$. In converting RAMs to circuits, it is convenient to assume that the addresses accessed by a RAM running in time $t(n)$ have length $O(\log t(n))$. In the next lemma, we show how to "compress" the address space of a RAM that runs in time $t(n)$, so that all address lengths are $O(\log t(n))$. The cost of this is an additional factor of $O(\log t(n))$ to the running time of the RAM.

Previous work on compressing the space used by a RAM was done by Slot and Van Emde Boas (1988). Using hashing techniques, they showed that if a RAM M uses d distinct addresses, it can be simulated by another RAM M' for which the largest address is numbered $O(d)$. Moreover, if the *space* used by a RAM is defined as the sum over all cells of the length of the maximum value in

every cell, then the space used by M' is within a constant factor of the space used by M. However, the time required by M' may be exponential in the time required by M. In contrast, our simulation guarantees that the time required by M' is within a logarithmic factor of the time required by M, but the cost to M' is $O(t(n))$ extra space.

LEMMA 2.1. *Let M be a RAM which runs in time $t(n)$ and space $s(n)$. Then there is a RAM M' that accepts the same language as M, runs in space $O(s(n)+t(n))$ and time $O(t(n)\log t(n))$, and accesses only cells whose addresses are $O(t(n))$.*

PROOF. Let M be a RAM with running time $t(n)$. We construct a RAM M' that simulates M, but which uses a table to store the addresses of cells referenced by M, together with the values at those addresses. The table can be stored in $O(t(n))$ contiguous cells. It is organized as a sequence $T_0, T_1, \ldots$ of balanced binary search trees. As our running time analysis will show, this organization is chosen to ensure that the cost of accessing small addresses is not too large (a single binary tree would not suffice to obtain the desired running time).

Each node of a tree in the table is a record consisting of a cell address, the value at that address and pointers to the nodes which are its children, if there are any. The first tree T_0 contains at most two nodes, containing the addresses 0 and 1. For $i \geq 1$, the ith binary tree T_i contains addresses referenced by M between $2^{2^{i-1}}$ and $2^{2^i} - 1$, inclusive. The number $2^{2^i} - 1$ is the *index* of the ith binary tree. For each tree there is also a *tree record*, which contains the index of the tree, a pointer to the root of the tree, if any, and a pointer to the next tree record, if any. The trees are built dynamically by M' as it is simulating M.

Suppose the value of cell R_N is needed for an instruction. This is located by M' as follows. First, N is compared with the indices of the binary trees in order, until the first index is found which is at least N. If no such tree exists, new, empty trees and their tree records are created in sequence until one exists. Let i be such that $2^{2^{i-1}} \leq N \leq 2^{2^i} - 1$. Next, the tree T_i is searched for N. If not found, a node with the address N is added to the tree with value 0. Then, the value of R_N can be retrieved or updated as specified by the instruction.

Space for the trees can be allocated dynamically as needed, so that all space used by the algorithm is stored in contiguous memory cells. The total number of cells needed to store the table is $O(t(n))$, since $O(t(n))$ distinct cells are accessed by M and there are only $O(\log t(n))$ tree records.

The running time of M' is the time to simulate the instructions of M, including searching and updating the trees, plus the time to build the empty trees. The time to create an empty tree is dominated by the cost of computing the tree indices. Let $2^{2^i}-1$ be the largest index computed. Then some address, say N, referenced by M is at least $2^{2^{i-1}}+1$. Since N is computed within time $t(n)$ by M, any individual index up to $2^{2^{i-1}}$ can be computed in time $O(t(n))$, by repeated additions. The time to then compute 2^{2^i} from $2^{2^{i-1}}$ is $O(2^{2i})$, since, using repeated additions, 2^{i-1} additions are needed, each involving numbers of length $O(2^i)$. Since $\log N \geq 2^{i-1}$, this time is $O(\log^2 N)$. Finally, since from Cook and Reckow (1973), $N \leq 2^{O(\sqrt{t(n)})}$, this is $O(t(n))$. Hence, an empty tree can be built in $O(t(n))$ time. Since there are $O(\log t(n))$ trees, the total time required to build the trees is $O(t(n)\log t(n))$.

We next consider the time for M' to simulate an instruction of M. It is straightforward to show that for instructions 1,2,3,6, and 7, this time is the time for M to execute the instruction, plus a constant. The same is true for the conditional part of instructions 8 and 9. Consider instructions 4 and 5. For these instructions, there is extra overhead to locate operands or to update the result of the instruction. The time to search for the tree in which address R_j is stored (not counting the time to create new trees, which we have already accounted for) is $O(\log t(n)\log R_j)$. This is because there are $O(\log t(n))$ trees in the table, and hence R_j is compared with $O(\log t(n))$ indices, each of length $O(\log R_j)$. The time to search for R_j in its tree is $O(\log t(n)\log R_j)$. This is because the search tree is balanced, thus ensuring that R_j is compared with $O(\log t(n))$ entries, and also all entries in the tree have address length $O(\log R_j)$. (Note that if the table were organized as a single binary tree, the cost could be $\Omega(\log^2 t(n))$—this is why we use a sequence of binary trees.) Similarly, the time to insert R_j in the tree is $O(\log t(n)\log R_j)$. It follows that the time for M' to execute instructions 4 or 5 is $\log R_{R_j} + O(\log t(n))\log R_j$. Clearly, this is within a factor of $O(\log t(n))$ of the time taken by M for these instructions.

Hence, the total time taken by M' is $O(t(n)\log t(n))$. □

We say a RAM is *restricted* if for all inputs x, and all k, the kth instruction executed on x (if any) is a function only of k and $|x|$. Note that the **if** statement is considered to be just one instruction (an instruction of this form is also known as a guarded command): this is needed for Lemma 2.2. Thus, the flow of control of the program of a restricted RAM is oblivious of the actual input. We next show that any RAM with running time $t(n)$ can be simulated by a restricted RAM with running time $O(t(n))$. Again, this restriction later simplifies our reduction from a RAM to a circuit in Theorem 4.1. The proof of the next

theorem is straightforward, using ideas similar to those in the main proofs of Ashcroft and Manna (1971) and Böhm and Jacobini (1966).

LEMMA 2.2. *Any RAM M with running time $t(n)$ can be simulated by a restricted RAM M' with running time $O(t(n))$. Moreover, the space and the maximum address of a cell accessed by M' are within a constant factor of the space and the maximum address of a cell accessed by M, respectively.*

PROOF. Let M be a RAM with running time $t(n)$. Without loss of generality, we assume that cells $R_1, \ldots, R_3$ are never used by M. We describe a restricted RAM M' that simulates M. The idea of the construction is that M' repeatedly "cycles" through the instructions of M, testing which one should actually be executed at a given step, and executing that instruction.

To implement this, the instructions of M are numbered in sequential order. M' consists of a loop that is executed at most $t(n)$ times. The loop consists of a sequence of segments, one per instruction of M. M' maintains counters *inst* and *newinst*, which store the number of the current and next instructions of M to be executed, respectively. The counter *inst* is initialized to the number of the first instruction and is updated to *newinst* at the end of the loop. A third variable, *test*, is used in each segment k to test if $inst = k$ and if so, to execute the kth instruction of M. The variables *inst, newinst* and *test* are stored in memory cells $R_1, \ldots, R_3$.

Suppose the kth instruction of M is *instruction* and is not a **goto** or an **if** instruction. Then the kth segment simply tests if *inst* has the value k and if so, executes the kth instruction of M. Then *newinst* is updated to be the next instruction. This is accomplished by the following code:

$test \leftarrow k$
$test \leftarrow test - inst$
if $test = 0$ **then** *instruction*
if $test = 0$ **then** $newinst \leftarrow k + 1$

In the case of a **goto** L instruction, the segment code is simplified by removing the first **if** statement and replacing the second by: **if** $test = 0$ **then** $newinst \leftarrow L$. Finally, suppose that the kth instruction is of the form **if** $R_i = 0$ **then** *instruction*. Then the third line of the above algorithm is replaced by the following two lines, which ensure that *instruction* is executed only if $R_i = 0$.

if $test = 0$ **then** $test \leftarrow R_i$
if $test = 0$ **then** *instruction*

The statement **if** $R_i > 0$ **then** *instruction* is handled in a similar fashion. In the kth iteration of the loop, the cost of the instructions is $O(1)$, plus the cost of the kth instruction of M that is executed. Therefore, the running time of M' is $O(t(n))$. □

2.2. Parallel RAMs and complexity classes. Our model of parallel computation is the PRAM model (see Fortune and Wiley 1978), which consists of a set of RAMs, called *processors*, plus an infinite number of *global* memory cells. An instruction of a processor can access its own memory cells or the global memory cells. There are many variations of the PRAM model, depending on how simultaneous access to a global memory location by more than one processor is handled. However, since the various PRAM models with a polynomial number of processors can simulate each other with $O(\log n)$ loss in time, the particular model does not affect our results. See Karp and Ramachandran (1988) or Greenlaw *et al.* (1994) for a survey of the various PRAM models. Note that the sequential running time of a problem on the RAM is within a constant factor of its parallel running time on the PRAM with $O(1)$ processors; thus, any speedup on the PRAM is due to parallelism and not to a change between models.

In what follows, we only consider PRAMs that use a number of processors that is polynomial in the input length (that is, the number of memory cells containing the bits of the input). With this assumption, we say that a language has *parallel running time* $T(n)$ if there is a PRAM accepting the language with running time $T(n)$. We say a language L has *parallel speedup* $s(n)$ if there is a PRAM accepting L with running time $t(n)/s(n)$, given that L has sequential running time $t(n)$. L has *polynomial speedup* if it has parallel speedup $\Omega(n^\epsilon)$ for some $\epsilon > 0$. We use the notation $\widetilde{O}(f(n))$ to denote $O(f(n)\log^{O(1)} f(n))$.

The class NC is the set of languages with parallel running time $\log^{O(1)} n$. A language L is NC-reducible to language L' if there is an NC-computable function f that is a many-one reduction from L to L'. See Greenlaw *et al.* (1994) and the references therein for an introduction to the class NC. We say f is *honest* if there exists k such that $|f(x)| \geq |x|^{1/k}$ for sufficiently large x.

3. Strict P-completeness

In this section, we define the notion of strict P-completeness. This is a refinement of the standard definition of P-completeness (see Greenlaw *et al.* 1994), which is that a problem L in P is P-complete if every problem in P is many-one NC-reducible to L. Roughly, a problem L in P is strictly $T(n)$-complete for P

if L has parallel running time within a polylog factor of $T(n)$ and furthermore, if this could be improved by a polynomial factor, then *all* problems in P with at least linear running time have polynomial parallel speedup. We now make this precise.

DEFINITION 3.1. *A language L in P is* at most $T(n)$-complete for P *if and only if for every language $L' \in P$, for every sequential RAM that accepts L' with running time $t'(n)$ where $t'(n) = \Omega(n)$ and $t'(n)$ is eventually non-decreasing, for every $\epsilon > 0$, there exists an honest, many-one NC reduction f from L' to L such that*

$$T(|f(x)|) = O(t'(|x|)|x|^{\epsilon}).$$

DEFINITION 3.2. *L is* strictly $T(n)$-complete for P *(or simply* strictly $T(n)$-complete*) if L is at most $T(n)$-complete for P and the parallel running time of L is $\widetilde{O}(T(n))$.*

We claim that if L is at most $T(n)$-complete for P, then if the parallel running time of L is a polynomial factor less than $T(n)$, every problem in P with at least linear sequential running time has polynomial speedup. This is evidence that parallel running time $\widetilde{O}(T(n))$ is the best one can expect for L. We prove this in the next lemma.

LEMMA 3.3. *Suppose that L is at most $T(n)$-complete for P. If the parallel running time of L is a polynomial factor less than $T(n)$, then every problem in P with sequential running time $t'(n)$ has polynomial speedup, where $t'(n) = \Omega(n)$ and $t'(n)$ is eventually non-decreasing.*

PROOF. Let L' be any problem in P with sequential running time $t'(n)$, where $t'(n)$ satisfies the hypothesis of the lemma. Let f be an honest, many-one, NC reduction from L' to L. Since f is honest, there is an integer k such that $|f(x)| \geq |x|^{1/k}$, for sufficiently large x. Suppose also that there is a parallel algorithm A for L that runs in time $O(T(n)n^{-2k\epsilon})$, for some $\epsilon > 0$. Consider the parallel algorithm for L' that on input x, computes $f(x)$ and runs the parallel algorithm A on $f(x)$. This algorithm has running time $O(T(|f(x)|)\ |f(x)|^{-2k\epsilon})$. This is $O(t'(|x|)\ |x|^{-\epsilon})$ since $T(|f(x)|) = O(t'(|x|)\ |x|^{\epsilon})$ and $|f(x)| \geq |x|^{1/k}$ for sufficiently large x. Hence there is a parallel algorithm for L' which achieves speedup $\Omega(n^{\epsilon})$ over the sequential algorithm with running time $t'(n)$. Hence L' has polynomial speedup. □

In Theorem 4.1, we show that the Square CVP is strictly $\sqrt{n}$-complete. The next lemma is useful in extending Theorem 4.1 to obtain bounds on the

parallel speedups of other problems. (In the following lemma, the definition of $s^{-1}(n)$ can be chosen in any reasonable way if s is not onto; for example, let $s^{-1}(n) = m$, where $s(m) = n'$ if n' is the largest number no greater than n in the range of s.)

LEMMA 3.4. *Suppose L is at most $T(n)$-complete and that there is an honest, many-one,* NC *reduction f from L to L'. Let $s(n)$ be the maximum of $|f(x)|$ over all instances x of L of size n. Suppose $T(n)$ and $s(n)$ are eventually non-decreasing functions. Then, L' is at most $T(s^{-1}(n))$-complete.*

PROOF. Let L'' be an arbitrary language in P with sequential running time $t''(n) = \Omega(n)$ where $t''(n)$ is eventually non-decreasing and let $\epsilon > 0$. To prove the lemma, we show that L'' is NC-reducible to L' by an honest function f' such that $T'(|f'(x)|) = O(t''(|x|)\ |x|^{\epsilon})$.

Since L is at most $T(n)$-complete, L'' is NC-reducible to L via an honest reduction f'' such that $T(|f''(x)|) = O(t''(|x|)|x|^{\epsilon})$. Let $f'(x) = f(f''(x))$. Then f' is an honest NC reduction from L'' to L'. Also,

$$T(s^{-1}(|f'(x)|)) = T(s^{-1}(|f(f''(x))|)) \leq T(s^{-1}(s(|f''(x)|)))$$

$$= T(|f''(x)|) = O(t''(|x|)|x|^{\epsilon}). \quad \square$$

4. The Square Circuit Value Problem

The Circuit Value Problem, CVP, denotes the set of all Boolean circuits together with an input to the circuit, such that the circuit evaluates to 1 on the input. This problem is well known to be complete for P (see Ladner 1975) and in fact many restrictions of the CVP are also P-complete. Greenlaw *et al.* (1994) proved that the CVP is P-complete even when accepted instances must be monotone, alternating, synchronous and with fan-in and fan-out 2. Here, monotone means that the gates are all **and** or **or** gates, alternating means that on any path of the circuit starting at an input gate, the gates alternate between **and** and **or** gates and synchronous means that the inputs for level i of the circuit are all outputs of level $i-1$.

We now define a *square* circuit to be one with all of the above properties and in addition, such that the number of gates at every level equals the depth. The *Square CVP* is the subset of CVP where the circuits are square.

The next theorem shows how to simulate RAMs with running time $t(n)$ by square circuits with depth $\widetilde{O}(t(n))$. Related work on simulating PRAMs

by circuits was done by Stockmeyer and Vishkin (1985), who showed that parallel time and number of processors for Concurrent Read, Concurrent Write PRAMS correspond to depth and size for unbounded fan-in circuits, where the time-depth correspondence is to within a constant factor and the processor-size correspondence is to within a polynomial. Our proof differs from theirs in that we are simulating a RAM rather than a PRAM and want a precise bound for the size of the circuit in terms of the running time of the RAM. Furthermore, our circuits have bounded fan-in and are synchronous.

THEOREM 4.1. *Any RAM that runs in time $t(n) = \Omega(n)$ can be simulated by a family of square circuits of depth $\tilde{O}(t(n))$. Furthermore, for a specific RAM and input, the corresponding circuit can be constructed in* NC *(actually, by a CREW PRAM with a polynomial number of processors which runs in $O(\log^2 n)$ time).*

PROOF. Given a RAM with running time $t(n)$, let M be a RAM that accepts the same language and has the following properties. There is a number $T = O(t(n) \log t(n))$ such that on inputs of length n,

(i) T is an upper bound on the maximum running time of M,

(ii) only cells $R_0, \ldots, R_T$ are accessed during a computation and

(iii) the total sum of the lengths of the non-zero cell values at any step of a computation is at most T.

Such a machine M exists with properties (i) and (ii) by Lemma 2.1, and with property (iii) because of the log cost assumption. Furthermore, we can assume that M is a restricted RAM, by Lemma 2.2.

We construct a circuit that simulates M on inputs of length n. We describe the construction of the circuit in stages. We first describe how to construct a circuit of the correct size and depth, and then describe the transformations needed to make the circuit square.

High level description. The circuit consists of T layers. The kth layer corresponds to the kth instruction of M that is executed on inputs of length n. Since M is restricted, this depends only on n and k. The output gates of the kth layer output the values of the cells $R_0, \ldots, R_T$ after the kth instruction is executed and are inputs to the $(k+1)$st layer. In addition, the $(k+1)$st layer may also have constant 0 and 1 inputs. One simple scheme would be to allow T output gates per cell, at each layer. However, this would require the circuit to have width $\Omega(t^2(n))$, which is too big.

We use T tuples of output gates, one per bit value of the cells. Each tuple contains $O(\log T)$ gates to record the bit position, $O(\log T)$ gates to record the cell number and one gate to record the value of the bit. Call these tuples the *cell tuples*. Tuples with all gates equal to 0 do not correspond to any bit value.

Constructing layers. A layer consists of a circuit for an instruction, preceded by *select* circuits that select the correct inputs for the instruction and an *update* circuit that updates the T edge tuples, based on the result of the instruction. We next describe each of these three circuits in more detail.

For all instructions of a RAM, there is a circuit of depth $O(\log t(n))$ and size $O(t(n))$ that computes the result of the instruction, if any, on inputs of length at most $t(n)$. The circuit is trivial for the instructions $R_i \leftarrow C$, **goto** L and **halt**. In the case of the indirect addressing instruction $R_i \leftarrow R_{R_j}$, the input to the circuit is R_{R_j}, so the circuit simply outputs its input. The circuit for addition and subtraction is based on the parallel prefix circuit of Ladner and Fischer (1980). The circuit for the **if** $R_i = 0$ **then** *instruction* contains a subcircuit which checks that $R_i = 0$ and another for *instruction*. There is an additional subcircuit which selects as the output either the output of the *instruction* circuit if $R_i = 0$, or a 0-valued output otherwise.

For instructions other than $R_i \leftarrow R_{R_j}$, the *select* circuit first selects the tuples recording bit values of the operands of the instruction, orders them and "pads" them on the left to T bits. For the instruction $R_i \leftarrow R_{R_j}$, *two* layers of the *select* circuit are needed. One layer selects the value R_j and the next layer selects the value R_{R_j}. The *select* circuit can be implemented in $\log^{O(1)} t(n)$ depth and $\widetilde{O}(t(n))$ size using a sorting circuit, for example, based on the sorting network of Batcher (1968) or Ajtai *et al.* (1983).

The *update* circuit receives as input both the result of the instruction and the bit values of all the cells with their labels. The *update* circuit "unpads" the result of the instruction and outputs these bits, correctly labeled. In the case of the instruction $R_{R_j} \leftarrow R_i$, the label is the value of R_j; otherwise, the label can be hard wired into the circuit. The *update* circuit also outputs, unchanged, the bit values of the cells that are not modified by the instruction, together with their labels. Again, a sorting circuit can be used to implement the *update* circuit in $\log^{O(1)} t(n)$ depth and $\widetilde{O}(t(n))$ size.

Achieving monotonicity, alternation, and restricted fan-in. We now have a circuit of the desired depth and size. To complete the proof, we show how this circuit can be transformed into a square circuit. The construction of Goldschlager (1977) can be used to make the circuit monotone. This construction doubles the size of the circuit and does not increase the depth.

Techniques of Greenlaw *et al.* (1994) can be used to ensure that the circuit is alternating and that each gate has fan-in and fan-out 2. These techniques can be applied to each layer independently so that the resulting circuit has the following structure.

In the first layer of the circuit, all inputs are connected to **or** gates. All output gates of every layer are **or** gates. An output gate of layer k is either not connected to anything, or the two edges from the gate are connected to an **and** gate at layer $k+1$. We refer to these **and** gates as inputs of layer $k+1$, in addition to the real inputs of the layer.

Synchronizing layers. Greenlaw *et al.* (1994) also describe how to make the resulting circuit synchronous, but their method at least squares the size of the circuit. We show that for the layered circuit of the above form, their technique can be modified to obtain a square, synchronous circuit of depth $\widetilde{O}(t(n))$. There are two main steps. First, each layer is converted into a synchronous circuit of depth $O(\log^{O(1)} t(n))$ and size $\widetilde{O}(t(n))$. This transformation of the layers increases the width of the layers, and hence duplicates the inputs and outputs of each layer. A problem caused by this is that the set of outputs of one layer may not equal the set of inputs to the next layer. The second main step, described in the next subsection, is to connect the synchronized layers. This is done by transforming in a synchronous fashion the set of outputs of one layer so that it equals the set of inputs of the next layer.

Following Greenlaw *et al.* (1994), a layer of depth d is made synchronous as follows. Construct $d/2$ copies of the circuit. Each copy can be viewed as having two levels. The **and** level consists of **and** gates and inputs; the **or** level consists of **or** gates. The edges between **and** levels and **or** levels are preserved within each copy. For all $l, 1 \leq l \leq d/2$, if **or** gate i is connected to **and** gate j in the layer, then gate i of the $(l-1)$st copy is connected to gate j of the lth copy. Input and output gates are handled as follows. An input gate at any copy is connected via a chain of alternating **and** and **or** gates to a duplicate of that input at the first level. Similarly, an output gate from any copy is connected via a chain of alternating **and** and **or** gates to the last level. The resulting layer is clearly synchronous and the properties of monotonicity, alternation, and restricted fan-in and fan-out are preserved by this transformation. It follows that there must be the same number of nodes at every level. Furthermore, the number of nodes at each level is $\widetilde{O}(t(n))$. This is because there is at most one node at each level for each gate of the original layer which is not an input or output gate, and at most d nodes at each level for each input and output gate of the original layer. The depth of the layer does not increase. The inputs to the synchronized layer are duplicates of the inputs to the original layer (both

constant inputs and the outputs of the previous layer), and the output gates of the synchronized layer are both duplicates of the output gates of the original layer and other arbitrary output gates, corresponding to other **or** gates in the original layer.

Connecting the synchronous layers. To complete the proof, we need to show how to transform the set of outputs of one layer into the set of inputs of the next layer in a synchronous fashion. By adding extra gates to some layers, we can assume that the number of gates at every level of every layer is equal. We can assume that the set of output gates of any layer includes the cell tuples and in addition, two gates with value 0 and 1 respectively. These two outputs can be produced by a synchronous layer simply by adding constant 0 and 1 gates as inputs and using **or** and **and** gates as identity gates to propagate these values to the output level of the layer. Call this set of outputs the *essential* outputs and the other outputs the *redundant* outputs. The set of inputs of layer $k+1$ can be produced by duplicating the essential outputs and replacing the redundant outputs of layer k with these duplicates.

To do this, first the edges from the redundant output gates are set to 0. This is achieved by a synchronous circuit of depth $O(\log t(n))$: the following "doubling" circuit. At the first **and** level of the circuit, the edges from one output gate are set to 0 by **and**-ing them with the outputs of the constant 0 gate. At the ith **and** level, there are 2^i 0-valued edges, which are **and**-ed with 2^i other redundant edges (or the remaining redundant edges if there are fewer than 2^i), to produce 2^{i+1} 0-valued edges. Output gates whose edges are unchanged at an **and** level simply pass their values on via an **and** gate used as an identity gate. At **or** levels, **or** gates are used as identity gates.

Essential outputs are duplicated in a similar fashion. An output gate g can be duplicated l times in depth $\lceil \log l \rceil$ by doubling the number of duplicates at each level (except possibly the last, if the number of duplicates is not a power of 2). In this part of the circuit, **and** gates are used as identity gates. This duplication can be done independently for all essential outputs in depth $O(\log t(n))$.

Since every gate of the circuit has fan-in and fan-out 2 and the circuit is synchronous, every level must have the same number of nodes. If this is less than the depth of the circuit, extra levels can be added to the circuit so that the depth equals the width. This completes the proof of Theorem 4.1. □

We can now show that the Square CVP is strictly $\sqrt{n}$-complete. Before giving the proof in Theorem 4.3, we need one technical lemma. The proof of this is due to J. Hoover.

LEMMA 4.2. *Let t be an eventually non-decreasing function such that $t(n) = \Omega(n)$ and $t(n) = n^{O(1)}$. For all $\delta > 0$, there is a rational number σ and a natural number n_0 such that*

1. *$t(n) \leq n^\sigma$ for $n \geq n_0$,*
2. *$n^\sigma = O(t(n)n^\delta)$.*

PROOF. Consider the real $\alpha \geq 0$ given by the following equation:

$$\alpha = \sup \left\{ k \;\middle|\; \lim_{n\to\infty} \frac{n^k}{t(n)} = 0 \right\}.$$

Such an α exists since $t(n) = \Omega(n)$, $t(n) = n^{O(1)}$ and t is eventually non-decreasing. For all $\epsilon > 0$, we have

$$\lim_{n\to\infty} \frac{n^{\alpha+\epsilon}}{t(n)} > 0,$$

and so there exists an n_ϵ such that $t(n) \leq n^{\alpha+\epsilon}$ for $n \geq n_\epsilon$.

Let σ be any rational number in the interval $[\alpha + \delta/4, \alpha + \delta/2]$. Then there exists n_0 such that $t(n) \leq n^\sigma$ for $n \geq n_0$ and we have condition 1 of the lemma.

Now, suppose that $0 < \alpha$ (if not, we are done immediately). Then, for some sufficiently small ϵ, $0 < \epsilon < \min\{\alpha, \delta/4\}$ we have

$$\lim_{n\to\infty} \frac{n^\sigma}{t(n)n^\delta} = \lim_{n\to\infty} \frac{n^{\alpha-\epsilon}n^{\sigma-\alpha+\epsilon}}{t(n)n^\delta} = 0.$$

Hence, $n^\sigma = O(t(n)n^\delta)$, and we have condition 2. □

THEOREM 4.3. *The Square CVP is strictly $\sqrt{n}$-complete.*

PROOF. Given a square circuit with n gates and an input for the circuit, there is a PRAM algorithm with running time $\widetilde{O}(\sqrt{n})$ that simply evaluates the gates of each row of the circuit in parallel. (On a Concurrent Read, Concurrent Write PRAM the time is $O(\sqrt{n})$; but on an Exclusive Read, Exclusive Write model an extra logarithmic factor is introduced).

We next show that the Square CVP is at most $\sqrt{n}$-complete. Let L' be a language in P and let $t'(n) = \Omega(n)$ be a sequential running time for L', with $t'(n)$ eventually non-decreasing. Let M' be a RAM that accepts any $x \in L'$ in time $t'(|x|)$. Let $T(n) = \sqrt{n}$.

For any $\epsilon > 0$, we show that there is an honest, many-one reduction f from L' to Square CVP such that $T(|f(x)|) = O(t'(|x|)\ |x|^\epsilon)$. By Lemma 4.2, there is a rational number σ and a natural number n_0 such that

1. $t(n) \leq n^\sigma$ for $n \geq n_0$,
2. $n^\sigma = O(t(n)n^\delta)$.

Since $\lceil n^\sigma \rceil$ is computable in NC, and $t'(n) = \Omega(n)$ implies $n^\sigma = \Omega(n)$, we can apply Theorem 4.1 to machine M' and input x and obtain a square circuit of depth $\widetilde{O}(|x|^\delta)$. For inputs $x \geq n_0$, this circuit correctly decides membership of x in L'.

Let f be the function that, given x, produces the corresponding square circuit. $|f(x)|$ is order the size of this circuit, and so $|f(x)| = \widetilde{O}(|x|^{2\sigma})$. By condition 1 above, and the lower bound on $t'(n)$, we conclude that f is honest. By condition 2 above, we get $|f(x)| = \widetilde{O}(t'(|x|)^2\ |x|^{2\epsilon})$.

Thus, $T(|f(x)|) = \widetilde{O}(t'(|x|)\ |x|^\epsilon)$ as required. □

5. Completeness results for other problems

We now apply Theorem 4.1 to prove strict completeness results for other problems. We first consider the problem of computing the expected cost of a Markov decision process, a central and well-studied problem in optimization theory. P-completeness results for Markov decision processes were obtained by Papadimitriou and Tsitsiklis (1987); we strengthen their results in the case of nonstationary finite horizon processes in Section 5.1. In Section 5.2, we describe a number of problems that are at most $\sqrt{n}$-complete for P, but which are not known to be strictly P-complete.

5.1. Markov decision processes. A Markov decision process consists of a set S of m states, including an initial state s_0. There is a finite set D of decisions and a probability transition function $p(s, s', d, t)$ which gives the probability of going from state s to s' at time t, given that decision d is made. The cost function $c(s, d, t)$ gives the cost of decision d at time t if the process is in state s. We consider the problem of minimizing the expected cost over T steps, starting at the initial state. The decision version of this problem, the *Nonstationary Finite Horizon Markov Decision Process Problem (NMDP)* is defined as follows: given an instance (S, s_0, D, p, c, T), is the expected cost equal to 0?

A sequential algorithm for this problem is based on dynamic programming (Howard 1960). Roughly, the algorithm fills in a table of size $T \times m$ row by row, where entry (t, s) of the table is the minimum expected cost when t is the finite horizon and s is the initial state. A table entry in row t can be

computed independently from the entries in row $t-1$. Thus these entries can be computed in parallel, and the computation of each entry can be parallelized so that the parallel running time is within a polylog factor of T. For example, if $T = \Theta(m)$ so that the input size $n = \Theta(m^2 T)$, then the sequential algorithm has a running time of $\Omega(n)$, whereas the parallel running time is $\widetilde{O}(n^{1/3})$.

Thus, the bottleneck in the above parallel algorithm for NMDP seems to be that the rows of the table must be computed in sequential order. We now show that under the assumption that $T = \Theta(m)$, NMDP is strictly $n^{1/3}$-complete for P. Informally, this implies that if there is a parallel algorithm for the NMDP with running time $O(T^{1-\epsilon})$ for some $\epsilon > 0$, then all problems in P have polynomial speedup.

THEOREM 5.1. *The Nonstationary Finite Horizon Markov Decision Process Problem where $T = \Theta(m)$ is strictly $n^{1/3}$-complete.*

PROOF. We reduce the Square CVP to NMDP. Our reduction is based on the reduction of Papadimitriou and Tsitsiklis (1987), but achieves an improvement by a factor of $\sqrt{n}$ on the number of states. Let C be a square circuit with n gates. Order the gates at each level of C from 1 to $\sqrt{n}$. We construct a nonstationary finite horizon Markov decision process M with $m = \sqrt{n} + 1$ states and let $T = m$. One state is a special *end* state q and the other m states correspond to the gates of C at any given level of the circuit. Thus, a pair $(s,t), 1 \le s \le m, 1 \le t < T$, corresponds to the sth gate in the circuit at level t, where the output gate is defined to be at level 1. The initial state is the number of the output gate of the circuit. There are two decisions, 1 and 2, associated with each state.

The probability transition function is defined as follows. $p(s,q,d,T) = 1$, for all s. For $1 \le t < T$, suppose that the sth gate at level t of the circuit has as inputs gates $(s_1, t+1)$ and $(s_2, t+1)$. If (s,t) is an **and** gate, then for $d = 1, 2$, $p(s, s_1, d, t) = p(s, s_2, d, t) = 1/2$. If (s,t) is an **or** gate, then $p(s, s_1, 1, t) = 1 = p(s, s_2, 2, t)$. All other probabilities are 0. The cost function $c(s,q,t) = 1$ if (s,t) is a false input gate. All other costs are 0.

The resulting process M has expected value 0 if and only if the circuit evaluates to 1; moreover the reduction is NC-computable and maps instances of the Square CVP of size n to instances of the NMDP of size $n^{3/2}$. Applying Lemma 3.4, it follows that NMDP is at most $n^{1/3}$-complete. □

5.2. Problems that are at most $\sqrt{n}$-complete. We next describe upper bounds on the parallel speedup obtainable for many other P-complete problems,

starting with the unrestricted Circuit Value Problem. The proof of the following corollary is immediate from Theorem 4.1.

COROLLARY 5.2. *The CVP is at most $\sqrt{n}$-complete for P, even when the circuits are restricted to be monotone, alternating and synchronous, with fan-in and fan-out 2.*

However, it is an open problem whether CVP is strictly $\sqrt{n}$-complete, since on the one hand, there is no known $\widetilde{O}(\sqrt{n})$ parallel algorithm for the CVP and on the other hand, it is not clear that a RAM which runs in time $O(t(n))$ can be simulated by a circuit family of size $O(t(n))$. Examples of other problems that are at most $\sqrt{n}$-complete are given in the following list. The completeness follows from Lemma 3.4 and the reductions cited below.

- *First Fit Decreasing Bin Packing (FFDBP).* Anderson, Mayr and Warmuth (1989) gave an NC-reduction from the Monotone CVP with fan-in and fan-out 2 to FFDBP. The reduction maps a circuit with n gates to an instance of FFDBP with $O(n)$ items whose weights are represented using $O(\log n)$ bits.
- *Ordered Depth First Search (ODFS).* Reif (1985) gave an NC-reduction from NOR CVP to ODFS that maps a circuit with n gates to a graph of size $O(n)$. There is a simple reduction from the Square CVP to NOR CVP that preserves the size of the circuit to within a constant factor (see Greenlaw *et al.* 1994).
- *Stack Breadth-First Search (SBFS).* Greenlaw (1988) gave an NC-reduction from the Monotone, Alternating, Synchronous CVP to SBFS. The same reduction maps an instance of the Square Circuit Value Problem with n gates to an instance of SBFS of size $O(n)$.
- *Unification.* Dwork, Kanellakis and Mitchell (1984) reduced the Monotone CVP to Unification, mapping a circuit of size n to an instance of Unification of size $O(n)$.
- *Lex First Maximal Independent Set (LFMIS).* Cook (1985) gave a linear-sized reduction from the Monotone CVP to LFMIS.

Finally, the *Lex First Maximal Clique (LFMC)* is an example of a problem for which there is a $\widetilde{O}(\sqrt{n})$ parallel algorithm, and the problem is at most $n^{1/4}$-complete. Cook (1985) gave a reduction from Monotone CVP to LFMC which

maps a circuit with n gates to a graph with $O(n)$ nodes but $\Omega(n^2)$ edges. There is a parallel algorithm that solves the problem in parallel time $\widetilde{O}((n+m)^{1/2})$. This algorithm maintains an ordered list of possible clique candidates, which are nodes connected to all nodes already in the clique. While this list is not empty, the smallest is removed and added to the clique, and the list is pruned of those candidates not adjacent to the newly added node. The list pruning can be performed in parallel for each list member in $\log^{O(1)} n$ time on a PRAM. Hence the total running time is within a polylog factor of the number of nodes in the clique. Clearly, the number of nodes in the LFMC is at most the square root of the number of edges in the graph, and hence is at most a square root of the input size.

6. Conclusions

We have introduced the notion of strict P-completeness, which refines previous work on P-completeness. We proved that the Square Circuit Value Problem and the Nonstationary Finite Horizon Markov Decision Process Problem are strictly $\sqrt{n}$-complete and $n^{1/3}$-complete, respectively. In doing this, we obtained improved reductions from the RAM model to the circuit model of computation. We also obtained limits on the possible speedups for many other P-complete problems; in particular, showing that the CVP is at most $\sqrt{n}$-complete.

We note that Definitions 3.1 and 3.2 of strict P-completeness can be extended in a straightforward way from many-one reductions to Turing reductions.

An interesting open problem is whether the CVP is strictly $\sqrt{n}$-complete, strictly n-complete, or somewhere in between. A related question is whether there are ways to prove limits on the parallel speedup that can be obtained for the CVP problem, other than by direct reduction from the RAM model. Finally, we have not severely restricted the number of processors in our PRAM model, since we allow any polynomial number of processors. Can our results be strengthened to obtain sharper limits if there is a stricter bound on the number of processors available?

Acknowledgements

Thanks to Jim Hoover for carefully reviewing and correcting a previous draft of this paper. The current definition of strict P-completeness incorporates many of his suggestions. Also, the proof of Lemma 4.2 is due to him. Thanks also to

Ray Greenlaw and an anonymous referee for several invaluable comments on the presentation.

This work was supported by NSF grant numbers CCR-9100886 and CCR-9257241, with matching funds provided by IBM Rochester and the AT&T Foundation.

References

M. Ajtai, J. Komlos, and E. Szemeredi, An $O(n \log n)$ sorting network. *Combinatorica* **3** (1983), 1–19.

R. Anderson, E. Mayr, and M. Warmuth, Parallel approximation schemes for bin packing. *Inform. and Comput.* **82** (1989), 262–277.

E. Ashcroft and Z. Manna, The translation of "go to" programs to "while" programs. In *Proc. 1971 IFIP Congress*, Amsterdam, The Netherlands, North Holland Publishing Company, **1** (1972), 250–255. Reprinted in *Classics in Software Engineering*, ed. E. N. Yourdan. Yourdan Press, N.Y., 1979.

K. E. Batcher, Sorting networks and their applications. In *Proc. AFIPS Spring Joint Summer Computer Conference*, 1968, 307–314.

Böhm and Jacopini, Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM* **9** (5) (1966), 366–371. Reprinted in *Classics in Software Engineering*, ed. E. N. Yourdan. Yourdan Press, N.Y., 1979.

S. A. Cook and R. A. Reckhow, Time bounded random access machines. *J. Comput. System Sci.* **7** (1973), 354–375.

S. A. Cook, A taxonomy of problems with fast parallel algorithms. *Inform. and Control* **64** (1985), 2–22.

C. Dwork, P. C. Kanellakis, and J. C. Mitchell, On the sequential nature of unification. *J. Logic Prog.* **1** (1984), 35–50.

S. Fortune and J. Wyllie, Parallelism in random access machines. In *Proc. 10th Ann. ACM Symp. Theor. Comput.*, 1978, 114–118.

L. M. Goldschlager, The monotone and planar circuit value problems are log space complete for P. *SIGACT News* **9** (1977), 25–29.

R. H. Greenlaw, A model classifying algorithms as inherently sequential. *Inform. and Comput.* **97** (1992), 133–149.

R. H. Greenlaw, J. Hoover, and W. L. Ruzzo, *Topics in Parallel Computation: A Guide to P-Completeness Theory*. Oxford University Press, 1994. See also *A compendium of problems complete for P*, Department of Computing Science Technical Report TR 91-11, University of Alberta, March 1991. Also available as Department of Computer Sciences Technical Report TR 91-14, University of New Hampshire, March 1991 and Department of Computer Science Technical Report TR 91-05-01, University of Washington, May 1991.

R. A. Howard, *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, 1960.

R. M. Karp and V. Ramachandran, Parallel algorithms for shared-memory machines. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, Chapter 7, ed. J. van Leeuwen. MIT Press/Elsevier, 1990, 869–941.

C. P. Kruskal, L. Rudolph, and M. Snir, A complexity theory of efficient parallel algorithms. *Theoret. Comput. Sci.* **71** (1990), 95–132.

R. E. Ladner, The circuit value problem is log space complete for P. *SIGACT News* **7** (1975), 18–20.

R. E. Ladner and M. J. Fischer, Parallel prefix computation. *J. Assoc. Comput. Mach.* **27** (1980), 831–838.

C. H. Papadimitriou and J. N. Tsitsiklis, The complexity of Markov decision processes. *Math. Oper. Res.* **12** (1987), 441–450.

J. Reif, Depth first search is inherently sequential. *Inform. Proc. Lett.* **20** (1985), 229–234.

R. A. Simons and J. S. Vitter, New classes for parallel complexity: a study of unification and other complete problems for P. *IEEE Trans. Comput.* **35** (1986), 403–418.

C. Slot and P. Van Emde Boas, The problem of space invariance for sequential machines. *Inform. and Comput.* **77** (1988), 93–122.

L. Snyder, Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science* **1** (1986), 289–317.

L. Stockmeyer and U. Vishkin, Simulation of parallel random access machines by circuits. *SIAM J. Comput.* **13** (1985), 862–874.

Manuscript received 15 June 1993

Anne Condon
Computer Sciences Department
University of Wisconsin at Madison
1210 West Dayton St.
Madison WI 53706 USA
condon@cs.wisc.edu