Algorithmica (1996) 15: 287–301

Algorithmica

Designing Checkers for Programs that Run in Parallel[1]

R. Rubinfeld[2]

Abstract. Program correctness for parallel programs is an even more problematic issue than for serial programs. We extend the theory of program result checking to parallel programs, and find general techniques for designing such result checkers that work for many basic problems in parallel computation. These result checkers are simple to program and are more efficient than the actual computation of the result. For example, sorting, multiplication, parity, the all-pairs shortest-path problem and majority all have constant depth result checkers, and the result checkers for all but the last problem use a linear number of processors. We show that there are P-complete problems (evaluating straight-line programs, linear programming) that have very fast, even constant depth, result checkers.

Key Words. Parallel algorithms, Program result checking.

1. Introduction.

Verifying a program to see if it is correct is a problem that every programmer has encountered. Even the seemingly simplest of programs can be full of hidden bugs, and in the age of massive software projects, this problem is becoming increasingly important. The complexity of programming parallel computers is even greater. A general theory of *result-checking* algorithms was given in [7]. This approach recognizes that proving programs correct is very difficult to do, and with this in mind, aims at the easier task of checking that a program is correct on any given input. This easier problem is not only feasible, but often yields result checkers that are much simpler than the original program and therefore less likely to contain bugs.

Many result checkers in the sequential model of computation have been found for various types of problems. However, a user is unlikely to be willing to use a sequential result checker to verify the correctness of a result produced by a fast parallel algorithm. In this paper we extend the program result-checking framework to the setting of parallel programs and find general techniques for designing such result checkers. For example, we find techniques for result-checking programs which compute certain types of functions that have the property that they can be computed "indirectly," by calling the program on another, related input. We also present techniques based on quickly reconstructing the computation of a simple sequential algorithm, on duality and on constant depth reducibility among problems. We find result checkers for many basic problems in parallel computation. The checking process is either faster or more efficient than the computation of the result. Many of these result checkers are rather straightforward, such as the result checkers for parallel prefix and straight-line programming. Others involve a more intricate design and more complex proofs, such as the result checkers for majority,

[1] This research was done while at the Computer Science Division, University of California, Berkeley, and the International Computer Science Institute, Berkeley, California. Supported in part by an IBM Graduate Fellowship and NSF Grant No. CCR 88-13632.

[2] Department of Computer Science, Cornell University, Ithaca, NY 14853, USA.

Received October 1, 1993; revised September 22, 1994. Communicated by A. Borodin.

unary to binary conversion, parity, and convex hull. In addition, the sequential versions of the parallel result checkers given for integer sorting and the all-pairs shortest-path problems are the first deterministic sequential result checkers for those problems. All of the examples in this paper are written for the arbitrary and priority CRCW PRAM models (see pp. 698–700 of [16] for definitions of models).

The difference in the complexity of solving a problem in parallel with a polynomial number of processors, as compared to the depth complexity of result checking a problem (again with a polynomial number of processors) is often very dramatic. For example, we show that there are P-complete problems (evaluating straight-line programs, linear programming) that have very fast, even constant depth, parallel result checkers. Integer GCD is not known to be in RNC, yet a logarithmic depth parallel result checker exists for it [1]. Maximum matching is not known to be in NC (though it is in RNC), and it has a deterministic NC result checker. Multiplication, parity, and majority all have lower bounds of $\Omega(\log n/\log\log n)$ depth [6] when computed with a polynomial number of processors, yet all have (completely different) constant depth result checkers.

2. The Parallel Program Result-Checking Model.

In this section we describe the extension of the program result-checking model proposed in [7] to result checking for parallel programs.

DEFINITION 2.1 (Probabilistic Parallel Program Result Checker). A probabilistic program result checker for f is a probabilistic oracle program R_f with oracle P, which is used to verify, *for any program* P that supposedly evaluates f, that P outputs the correct answer on a given input in the following sense. On a given input x and confidence parameter α, R_f^P has the following properties:

1. If $P(x) \neq f(x)$, then R_f^P outputs "FAIL" (with probability $\geq 1-\alpha$).[3]
2. If P is a correct program for every input then R_f^P outputs "PASS" (with probability $\geq 1-\alpha$).

Furthermore, the result checker may only use a polynomial number of processors.

The parallel result checker is allowed to call the program as many times as desired at each parallel step but is only allowed to access P as a black-box oracle.

Note that if $P(x) = f(x)$ but P is faulty on other inputs, then R_f^P may output either "FAIL" or "PASS."

We refer to the parallel running time of the checker, not including the running time of the calls to the program, as the *checking time* (note that the running time of the checker may be an expected run time, even when the program being checked is deterministic). We refer to the parallel running time of the checker including the running time of the calls to the program as the *total time*. We make the same conventions with respect to the number of processors. When describing the total running time of a result checker,

[3] The probabilities are with respect to a source of truly random independent bits available to the result checker, and *not* with respect to any assumptions about the input distribution.

we use $D(n)$ to refer to the total time of the program running on an input of size n, and $N(n)$ to refer to the total number of processors used by the program when running on an input of size n. In both cases we ignore the dependence on the confidence parameter α. In all of the examples, the result checker first calls the program on the input which is being checked.

The problem remains of determining the correctness of the result checker. In the sequential setting, [7] suggests that instead, the result checker should be forced to be quantifiably "different" than any program for f by limiting the checking time to be less than that of the fastest correct program known for computing the function. In fact, the result checker can be forced to be "different" by restricting any computational resource. We consider analogous notions of "different" for parallel result checkers. Suppose that any parallel program which computes the function f using a polynomial number of processors requires at least $d(n)$ depth on inputs of size n. Suppose the number of processors required when computing f in depth $d(n)$ is $p(n)$. We say that R_f is *quantifiably different* if (i) the checking time is $o(d(n))$ depth *or* (ii) if the checking time is $O(d(n))$ and simultaneously the checking number of processors is $o(p(n))$ on the same model of parallel computation. The latter option is motivated from a more practical perspective, since checking algorithms satisfying (ii) do not require the existence of extra processors in order to run. The result checkers described in this work are all quantifiably different. So far, most of the result checkers found which are quantifiably different seem to be simpler than any program for the function as well.

Another possible variant for the notion of quantifiably different is as follows: Let $d'(n) > d(n)$. Suppose that no program can be adapted to run in time $d'(n)$ using $O(p'(n))$ processors (by Brent's theorem, programs running in $d(n)$ time and $p(n)$ processors can be adapted to run in time $d'(n)$ using $p'(n)$ processors when $p'(n)$ is $\geq p(n)(d'(n)/d(n))$). A checker is quantifiably different if its running time is $O(d'(n))$ using $O(p'(n))$ processors. This particular notion has not yet been used to construct a checker, but may be of practical interest.

In the most straightforward applications of checking, whenever the program is executed the result checker is also executed. Thus, it is critical that the overhead cost of running the result checker does not neutralize the benefit of knowing that the output is correct (or the knowledge that the program is faulty). All of the result checkers in this paper call the program at most once on any computation path, so the total depth is *big oh* of the depth of the program being checked. Many of the result checkers have the property that the total number of processors used is *big oh* of the number of processors used by the program (e.g., sorting, parity, convex hull).

3. Computability by Random Inputs.

In [8] it is shown that result checkers can be designed for many functions that have the property of random self-reducibility—that the function can be computed by computing the function on one or more "random" instances. We show that often the property that a function can be computed by computing the function on one or more "almost-random" instances can also be utilized in designing a result checker.

We concentrate on symmetric functions—functions on n bits whose output depends only on the *number* of ones in the input. Thus, the value of the function can be computed indirectly by computing the function on a "shuffle" (random permutation) of the input bits. However, the techniques in this section are applicable to other functions as well. For example, the running time of the sequential checker for the matrix rank function given in [8] can be dramatically improved using the technique given in Section 3.2.

The techniques in this section are based on testing the program on random inputs for which the answer is known, and then verifying that the program's answer on the particular input being checked is consistent with the program's answer on most other inputs. All techniques require only a $\log^* n$ *additive* factor overhead in the depth. (This can be made constant by using an extra $\log n$ multiplicative factor of processors). The first technique works for any symmetric function, but uses a multiplicative factor of n extra processors. The technique described in the second section works for a certain class of symmetric functions, and does not use any extra processors. It is an open question to determine whether there is a general technique to check any symmetric function without using extra processors. The third technique works for certain types of random self-reducible functions, and does not require extra processors.

3.1. *Any Symmetric Function on n Bits.* We give a result checker for any symmetric function, where the function is specified by a table of values $t_0, \dots, t_n$, such that t_i is the output of the symmetric function when exactly i of the input bits are ones:

Input: A list of input bits $\hat{a} = a_1, a_2, \dots, a_n$, a table of values $t_0, \dots, t_n$.
Output: $b = t_l$ where $l = \sum_{1 \le j \le n} a_j$.

The majority, exactly l, and parity functions are all examples of symmetric functions. As mentioned before, [6] show that $\Omega(\log n / \log \log n)$ depth is required to compute these functions. For these and other examples, no table is needed as input because each t_i can be computed in constant depth by the result checker.

Let P be the program that supposedly computes the symmetric function. P is checked by partitioning the inputs of size n into $n + 1$ equivalence classes, where all inputs in a particular equivalence class contain the same number of ones. Intuitively, the result checker verifies that P is correct on more than half of the members of each equivalence class, and that the answer of P on the input in question is consistent with more than half of the members within its own equivalence class. Therefore, even if the result checker cannot determine which equivalence class the input is in, it can verify that the answer of P on the input is correct. In the result-checking algorithm, several random permutations of the input bits are made; [23] provides a way of doing this in $O(\log^* n)$ depth with linear processors (and $O(\log^* n \log \log^* n)$ depth with an optimal number of processors). They also show how to do it in $O(1)$ depth and $O(n \log n)$ processors.

Result-Checking Algorithm

```
$k \leftarrow \log(1/\alpha)$
$b \leftarrow P(\hat{a})$
In parallel, compute $k$ random permutations $\pi_1, \dots, \pi_k$ of $\{1, \dots, n\}$
Phase 1: (Consistency with our input)
   In parallel, for $i = 1, \dots k$
      If $P(\pi_i(\hat{a})) \ne b$, then output "FAIL" and halt
Phase 2: (Testing Correctness of most inputs)
```

```
    In parallel, for $j = 0, \dots, n$
      In parallel, for $i = 1, \dots, k$
        If $P(\pi_i(1^j 0^{n-j})) \neq t_j$, then output "FAIL" and halt.
  Output "PASS."
```

PROOF OF CORRECTNESS. Clearly, if P is correct on all inputs, the result checker will output "PASS." Assume that P is incorrect on input $\hat{a}$, we show that the result checker outputs "FAIL" with probability $\geq 1 - \alpha$. Let l be the number of ones in $\hat{a}$. Suppose that P is correct (and consequently differs from the output on $\hat{a}$) on $\geq 1/2$ the inputs with l ones. Then, with probability $\geq 1 - \alpha$, an input that is inconsistent with $\hat{a}$ is found in Phase 1. Suppose that the program errs on $\geq 1/2$ the inputs of size n with l ones. Then, with probability $\geq 1 - \alpha$, the lth group of processors in Phase 2 finds that the program is buggy. Notice that by this argument the same k permutations can be used in Phase 1, and by every group of processors in Phase 2.

Running Time. The checking time is $O(\log^* n)$ and checking number of processors is $O(n^2)$. The total time is $O(\log^* n + D(n))$ and the total number of processors is $O(nN(n))$. □

3.2. *Special Symmetric Functions.* A factor of n in the number of processors can be saved when the symmetric function f is of a special type: Let $t_0, \dots, t_n$ be the input table for problems of size n and let $t'_0, \dots, t'_{2n}$ be the input table for problems of size $2n$. We say that f is of this special type if there is an easily computable function $g(b, j)$ such that if $t'_i = b$, then $t_{i-j} = g(b, j)$.

Examples of such functions are parity, where $g(b, j) = b \oplus (j \bmod 2)$, and the unary to binary conversion function, where $g(b, j) = b - j$.

Result-Checking Algorithm

```
  $k \leftarrow \log(1/\alpha)$
  $b \leftarrow P(\hat{a})$
  In parallel, compute $k$ random permutations $\pi_1, \dots, \pi_k$ of $\{1, \dots, 2n\}$
  Phase 1: (Consistency with our input)
    In parallel, for $i = 1, \dots, k$:
      Uniformly and randomly pick $j \in [0, \dots, n]$
      Let $s$ be the string $a_1, \dots, a_n, 1^j 0^{n-j}$
      $s' \leftarrow \pi_i(s)$
      If $b \neq g(P(s'), j)$, output "FAIL" and halt.
  Phase 2: (Testing Correctness of most inputs)
    In parallel, for $i = 1, \dots, k/\log(4/3)$:
      Uniformly and randomly pick $j \in [0, \dots, 2n]$
      Create the string $s = 1^j 0^{2n-j}$
      $s' \leftarrow \pi_i(s)$
      If $P(s') \neq t'_j$, output "FAIL" and halt.
  Output "PASS."
```

PROOF OF CORRECTNESS. If P is correct on all inputs, then clearly the result checker outputs "PASS." Let l be the number of ones in $\hat{a}$ and suppose $b = P(\hat{a}) \neq t_l$. Let $\mathcal{D}$ be the probability distribution defined by (j, r), where j is chosen uniformly at random in $[0, \ldots, 2n]$ and r is a random string of length $2n$ with j ones. Let $\mathcal{D}'$ be the probability distribution defined by (j, r), where j is chosen uniformly at random in $[0, \ldots, n]$ and r is a random string of length $2n$ with $j + l$ ones. Let p be the probability that $P(r) \neq t'_j$ when (j, r) is chosen according to $\mathcal{D}$.

If $p \geq 1/4$, then each execution of the loop in Phase 2 outputs "FAIL" and halts with probability at least 1/4. Thus, the output is "FAIL" with probability at least $1 - \alpha$.

Now consider the case where $p \leq 1/4$. Let s' be a string of length $2n$ with $(l + j)$ ones. Each string (j, r) that is generated by $\mathcal{D}$ with probability q, is generated by $\mathcal{D}'$ with probability 0 or $2q$. Therefore, if $p \leq 1/4$, then $\Pr[P(s') \neq t'_{l+j}] \leq 1/2$ when (j, s') is chosen according to $\mathcal{D}'$. By the properties of g, if $P(s') = t'_{l+j}$, then $g(P(s'), j) = t_l$. These two facts imply that $\Pr[g(P(s'), j) = t_l] \geq 1/2$ in each execution of the loop in Phase 1, and thus, since $b \neq t_l$, $\Pr[g(P(s'), j) \neq b] \geq 1/2$ in each execution of the loop in Phase 1, in which case the output is "FAIL." Thus, the output is "FAIL" with probability at least $1 - \alpha$.

Running Time. The checking time is $O(\log^* n)$ parallel steps and the checking number of processors is $O(n)$. The total time is $O(\log^* n + D(n))$ and the total number of processors is $O(n + N(n))$. □

3.3. *Randomly Self-Reducible, Linear, and Smaller Self-Reducible Problems.* If the program computes a function which is randomly self-reducible and either has the linearity property or is self-reducible to smaller inputs (see [8] for a definition), the general techniques described in [8] can be parallelized. This gives constant depth efficient result checkers for checking numerical problems such as integer multiplication, integer division, mod, modular multiplication, modular exponentiation, polynomial multiplication, squaring, and matrix multiplication. The technique can also be used to give a result checker for parity that uses $O(1)$ checking time and $O(n)$ checking number of processors.

4. Consistency.

Many problems have linear time sequential algorithms that are extremely simple and even possible to prove correct with formal verification methods. However, it is often the case that any parallel algorithm P for the same problem is necessarily radically different and more complex. Intuitively, a typical parallel result checker developed in this section calls P to reconstruct the computation steps of the extremely simple sequential algorithm, and then verifies the consistency between adjacent steps of the computation. This can be done very quickly, and independently of the algorithm actually used by P. This simple idea gives deterministic parallel result checkers for a number of problems. Many problems have result checkers that do not need any additional calls to P. Others require several calls to P. An important future direction of research is to reduce the overhead for these problems.

The *prefix sums* problem takes as input a list of elements $a_1, a_2, \ldots, a_n$, and outputs $(b_1, b_2, \ldots, b_n)$, where $b_i = a_1 \circ a_2 \circ a_3 \circ \cdots \circ a_i$ for an associative binary operator $\circ$. We assume that $\circ$ can be computed correctly by one processor in constant time. In order to verify that P computes the correct result, in parallel for $1 \leq i \leq n-1$, processor i checks that $b_i \circ a_{i+1} = b_{i+1}$. The checking time is $O(1)$ and the checking number of processors is n. The total depth is $O(D(n))$ with $O(N(n)+n)$ total processors. Note that the result checker makes no additional calls to P. A small variant of this result checker works for the *list-ranking problem* as well in the same time and with the same number of processors.

The *sum* problem is similar to prefix sums, except that only b_n is output, and thus it is harder to check. The intermediate prefix answers $b_1, \ldots, b_{n-1}$ can be reconstructed as follows: In parallel for $1 \leq i \leq n$, group i of processors calls the program to compute $b_i = P(a_1, a_2, \ldots, a_i)$. Then processor i verifies that $b_i \circ a_{i+1} = b_{i+1}$. The checking time is $O(1)$ and the checking number of processors is $O(n^2)$. The total depth is $O(1)+D(n)$ with $O(n \times N(n))$ total processors.

The ideas in this result checker can be used for various problems, including *parity*, *addition of n numbers*, and can be modified to work for *straight-line programming* (when the variables are each set only once) and the expression evaluation problem. When the variables can be set more than once, the checking time of straight-line programming is $O(\log n)$ using sorting.

A result checker for *integer multiplication* can also be constructed using this idea, where the input is two n-bit numbers a, b and the output is $a \times b$. The result checker algorithm is as follows: In parallel for $1 \leq i \leq n$, the ith group of n processors asks the program to multiply a by the last i bits of b to get r_i. If the ith least-significant bit of b is a zero, then the result checker verifies that $r_i = r_{i-1}$, otherwise the result checker verifies that $r_i - r_{i-1} = a \times 2^i$. The checking time is $O(1)$, and the checking number of processors is $n \times A(n)$, where $A(n)$ is the number of processors required to do addition in constant depth. The best algorithm for multiplication takes $O(\log n/\log\log n)$ time and $O(n^{1+\varepsilon})$ $(\varepsilon > 0)$ operations, by combining [27] with [10]. The total time is $O(D(n))$ with $O(n \times A(n) + n \times N(n))$ total processors.

Because of the following known results, all the checkers presented in this section are quantifiably different. When the input consists of integers, the best known algorithm for prefix sums uses $O(n/\log n)$ processors and $O(\log n/\log\log n)$ depth [11]. Any algorithm using only a polynomial number of processors for prefix sums, sum, parity, and integer multiplication provably requires $\Omega(\log n/\log\log n)$ depth [6]. Straight-line programming is P-complete.

4.1. *Problems that Can Be Solved Using Dynamic Programming.* Dick Karp has pointed out that the basic technique described in this section can be used to check any problem that can be solved sequentially using dynamic programming, regardless of the algorithm used by the program. By dynamic programming, we mean that there is some polynomial algorithm that computes the function on the whole set of inputs by evaluating the *same* function on smaller sets of inputs and somehow combining the results. This usually involves writing out the function on smaller sets of inputs in the form of a table. The idea behind the result checker is to call the program on *each* subproblem in parallel to fill in the table, and then verify that the entries of the table are consistent with each other. In most cases this combination of results involves finding the minimum or maximum of a set of numbers. Since the minimum and maximum function can be computed in constant time, the checking time is constant.

The following is an example:

LONGEST COMMON SUBSEQUENCE

Input: Two strings $x = x_1x_2x_3 \cdots x_n$ and $y = y_1y_2y_3 \cdots y_n$.
Output: The length of the longest common subsequence of x and y.

Let $\mathrm{lcs}(l, k)$ denote the length of the longest common subsequence of $x_lx_{l+1} \cdots x_n$ and $y_ky_{k+1} \cdots y_n$. Then the sequential dynamic programming algorithm used to solve the longest common subsequence problem builds up the table as follows: if $x_l = y_k$, then $\mathrm{lcs}(l, k) = 1 + \mathrm{lcs}(l + 1, k + 1)$, otherwise $\mathrm{lcs}(l, k) = \max\{\mathrm{lcs}(l, k + 1), \mathrm{lcs}(l + 1, k)\}$.

```
Do for all $1 \leq l \leq n$
  Do for all $1 \leq k \leq n$
    $s_{lk} \leftarrow P(x_l \cdots x_n, y_k \cdots y_n)$
    Verify consistency:
      If $x_l = y_k$ verify that $s_{lk} = 1 + s_{l+1,k+1}$
      else verify that $s_{lk} = \max\{s_{l,k+1}, s_{l+1,k}\}$
If any of these verifications fail, then output "FAIL" else output "PASS"
```

The checking time is $O(1)$ and the checking number of processors is $O(n^3)$. The total running time is $O(1 + D(n))$ with $O(n^3 + n^2 \times N(n))$ total processors.

4.2. *All-Pairs Shortest Path and Depth-First/Breadth-First Search Trees*

Input: $n \times n$ adjacency matrix A, with a nonnegative weight for each edge.
Output: Matrix *Dist* specifying length of shortest path between every pair of nodes.

Result-Checking Algorithm

```
Do in parallel for each entry $D(u, v)$
  (1) check that $Dist(u, v) \leq A(u, v)$
  (2) check that for all $w$ that are neighbors of $v$, $Dist(u, w) + A(w, v) \geq Dist(u, v)$
  (3) check that $\exists w$ neighbor of $v$ such that $Dist(u, w) + A(w, v) = Dist(u, v)$
If any of these checks fail, then output "FAIL" else output "PASS"
```

PROOF OF CORRECTNESS. It is clear that if the program is correct, the result checker will output "PASS." Suppose that the result checker outputs "PASS." Let $d(u, v)$ denote the correct shortest distance between u and v. We want to show that for all pairs (u, v), $Dist(u, v) = d(u, v)$.

Suppose for contradiction that there are nodes u, v such that $Dist(u, v) < d(u, v)$. Let u, v be nodes with $Dist(u, v) < d(u, v)$ such that v has the smallest possible index. Then, because of step 3, there must be a w such that $Dist(u, w) < d(u, w)$ and w has smaller index that v. Therefore, for all u, v we have that $Dist(u, v) \geq d(u, v)$.

We will show by induction on the number of intermediate nodes along a shortest path between a pair of nodes that $Dist(u, v) = d(u, v)$.

Basis. The number of intermediate nodes visited when taking the shortest path from u to v is zero (edge uv is the shortest path). Step 1 guarantees that $Dist(u, v) \leq A(u, v) = d(u, v)$.

Induction Step. Suppose that $Dist(u, v) = d(u, v)$ for all pairs (u, v) where there is a shortest path from u to v that visits i intermediate nodes. Consider pair (u, v) where there is a shortest path from u to v with $i + 1$ intermediate nodes, and let w be the last node along this path. Then, step 2 verifies that $Dist(u, w) + A(w, v) \geq Dist(u, v)$. We know $d(u, w) + A(w, v) = d(u, v)$. By the induction hypothesis, since there is a shortest path between u and w of length i, $Dist(u, w) = d(u, w)$. Thus $Dist(u, v) \leq A(w, v) + d(u, w) = d(u, v)$ and so $Dist(u, v) = d(u, v)$.

Running Time. The checking time is $O(1)$ and the checking number of processors is $O(n^3)$. The total time is $D(n) + O(1)$ with $O(n^3) + N(n)$ total processors. Note that the result checker makes no extra calls. □

Yossi Matias has shown that this idea can also be applied to checking programs that solve the problems of constructing depth-first and breadth-first search trees:

In the former problem the input is an undirected graph G (with n nodes and m edges) and some node r in G. The output is a rooted tree T which can be obtained by performing a depth-first search on G, starting from r. No "efficient" parallel algorithm for this problem is known. It was shown to be in RNC by Aggarwal and Anderson [2]. The checker first confirms that T is a tree, by checking that T is connected and has $n - 1$ edges. Then the checker confirms that all nontree edges (v, u) are backedges; i.e., that either v is an ancestor of u or u is an ancestor of v. Connectivity can be determined in $O(\log n)$ expected time and $O((m + n)/\log n)$ processors using the techniques of [13]. Finding lowest common ancestors for all edges can be done in $O(\log n)$ time using $O(n/\log n)$ processors [26].

In the problem of constructing a breadth-first search tree, the input is an undirected graph G and some node r in G. The output is a rooted (directed) tree T which can be obtained by performing a breadth-first search on G, starting from r. This is equivalent to the single source (unweighted) shortest-path problem. While in sequential computation this problem is easier than the all-pairs shortest-path problem, it is not known to be the case in parallel. The output may or may not include the distances of these shortest paths. In fact we discuss three different problems:

(1) Both the BFS tree and the distances are computed: The checker verifies that $d(r) = 0$. For each edge (v, u) in T the checker verifies that $d(v) = d(u) + 1$. For each node $v \neq r$ in T, the checker verifies that there is exactly one edge directed into v.
(2) Only distances are computed: The checker verifies that $d(r) = 0$. For each node v, the checker verifies that, for exactly one neighbor u, $d(v) = d(u) + 1$ (unless $v = r$), and that there is no neighbor w for which $d(w) > d(v) + 1$.
(3) Only a BFS tree is computed: The checker verifies that T is a tree. For each node v, the checker computes the level of v in T. For each edge (v, u), the checker confirms that $|level(v) - level(u)| \leq 1$.

The checking/total depth and checking/total number of processors in each case are:

(1) $O(1)$ time and $O(n)$ processors.
(2) $O(1)$ time and $O(m)$ processors.
(3) $O(\log n)$ time and $O(n)$ operations.

5. Sorting and Computational Geometry

5.1. *Sorting.* Consider the problem of sorting integers with the following specifications:

Input: A set of integers $X = \{x_1, x_2, \dots, x_n\}$ (not necessarily distinct).
Output: The elements of X in sorted order: i.e., a list $y_1 \le y_2 \le \cdots \le y_n$ such that $Y = \{y_1, \dots, y_n\}$ is equal to X.

In the algebraic decision-tree model and on the comparison-tree model, sorting requires $\Omega(n \log n)$ time. Although there are faster sequential algorithms for sorting small integers, there are no linear-time integer-sorting algorithms.

The result checker must verify that the output is in sorted order, and that the set of elements in the input list is the same as the set of elements in the output list. The first task is quite easy, but the second task is nontrivial, and, on the algebraic decision-tree model, is as difficult a task as sorting. In [7] there are randomized algorithms for verifying that $X = Y$ which use hashing and run in $O(n)$ time. We present a deterministic algorithm which checks sorting in $O(1)$ parallel time and $O(n)$ processors. This algorithm is the first deterministic sequential result checker for sorting that runs in $O(n)$ time.

Checker Algorithm. (For simplicity, assume that n is a power of 2.)

```
$Y \leftarrow P(X)$
Do in parallel for $1 \le i \le n$
    Append $\log n$ bits to the binary representation of the $i$th input indicating
      its location in the input list, i.e., $x'_i \leftarrow (x_i) \times n + i$. (Note that this
      does not affect the ordering of the elements.)
Let $X' = \{x'_1, \dots, x'_n\}$.
$Y' \leftarrow P(X')$
Do in parallel for $1 \le i \le n$
    Let $j$ be the last $\log n$ bits of $y'_i$: $j \leftarrow y'_i \bmod n$.
    Verify that $x'_j = y'_i$.
    $A[j] \leftarrow i$
    Verify that $A[j] = i$
Let $Y'' = \{y'_1 \operatorname{div} n, \dots, y'_n \operatorname{div} n\}$.
Verify that $Y$ is in sorted order, $|Y| = n$, and that $Y = Y''$.
If any verification fails, output "FAIL," else output "PASS."
```

5.2. *Planar Convex Hull*

Input: A list of points with their coordinates in R^2, labeled by their location in the input list: $(1, x_1, y_1), (2, x_2, y_2), \dots, (n, x_n, y_n)$.
Output: A description of the boundary of the convex hull. This description will be a list of vertices of the convex hull in counterclockwise order around the hull.
Model of Computation: CRCW PRAM in which arithmetic operations $(+, -, \cdot, /)$ on real numbers can be performed in one step.

The best algorithm for this problem in [3] runs in $O(\log n)$ depth and uses $O(n)$ processors.

The following algorithm result checks planar convex hull using a constant checking time, but uses many processors:

Result-Checking Algorithm. For each edge on the convex hull, n processors will be assigned to verify that all of the input points are on the same (correct) side of the edge. This can be done in constant parallel time with $O(n^2)$ processors.

The following algorithm result checks planar convex hull in a way that is is more efficient with processors.

Result-Checking Algorithm. This result checker is a parallel implementation of the sequential result checker of [15]. The result checker must verify that the polygon described in the output is simple and convex. This is done by assigning a processor to each vertex of the hull in order to verify that a left turn is made by the two edges adjacent to this vertex. Each processor determines whether a change in the x-direction of the walk is made. The processors then verify that a change in the x-direction of the walk is made at only two vertices. Next, the result checker must verify that all of the points not said to be on the hull are really inside the boundary. For each point not on the hull, it finds a "proof" that it is indeed inside the boundary. This proof will consist of three points in the input set whose convex combination contains the nonhull point. Suppose the convex hull were triangulated by drawing a line from the leftmost vertex to every other vertex on the convex hull. For each point not on the hull, the three points found that contain it will be the points on the triangle surrounding the nonhull point. To find these points, the result checker uses the program to sort the input points by angle around the leftmost point q (by transforming the input points by $(i, x_i, y_i) \rightarrow (i, \theta_i, \theta_i^2)$). The hull points in the sorted list are then marked. An easy modification of the parallel-prefix algorithm can be used twice in order to find, for each point p inside the hull, the closest convex-hull point such that its angle is smaller/bigger than that of p. Call these two points a and b. Then the checker verifies that triangle (q, a, b) contains p.

Running Time. The checking time is $O(\log n)$ and the checking number of processors is $O(n/\log n)$. The total time is $O(\log n + D(n))$ with $O(n/\log n + P(n))$ total processors.

5.3. *Three-Dimensional Convex Hull*

Input: A list of points with their coordinates in R^3 (in general position).
Output: A description of the boundary of the convex hull (vertices, edges, and faces). Without loss of generality, assume that the boundary is triangulated. For each vertex, the faces adjacent to it will be given in an order such that consecutive faces are adjacent. This will also induce an ordering on the edges, and the description will output the edges around each vertex in this order. Note that the number of edges and faces on a convex polyhedron is linear in the number of vertices of the polyhedron.
Model of Computation: CRCW PRAM in which arithmetic operations on real numbers can be performed in one step.

The best known parallel algorithm from the three-dimensional convex hull mentioned in [5] requires $O(\log n)$ time with $O(n^{1+\alpha})$ processors. The following result-checking algorithm uses a constant checking time, but uses many processors.

Result-Checking Algorithm. For each face on the convex hull, n processors will be assigned to verify that all of the input points are on the same (correct) side of the face. Since there are only $O(n)$ faces on a convex hull, this can be done in constant parallel time with $O(n^2)$ processors.

The following result-checking algorithm uses only $O(n)$ processors:

Efficient Checker. The result checker must verify that the polyhedron described in the output is simple and convex. This is done by checking that the polyhedron is locally convex at each point on the polyhedron. Though not enough in two dimensions, in three dimensions this is enough to show that the polyhedron is convex since any three-dimensional polyhedron which is locally convex at each point on the surface must also be globally convex (see [28]).

We describe how to check that the polyhedron is locally convex: Since the points on the interior of the faces are locally convex, the only points that must be checked are the points along the edges and the vertices of the polyhedron. Since the faces are ordered such that consecutive faces are adjacent, a processor can be assigned to each pair of consecutive faces (or three consecutive edges) in order to make sure that they are making convex turns. This can be done in $O(1)$ time and $O(n)$ processors. Checking that the vertices are locally convex reduces to several two-dimensional convex-hull problems with total size $O(n)$: For each hull vertex v, consider a plane that separates v from its neighbors on the polyhedron (if the polyhedron is truly convex, then this plane should separate v from all other vertices on the hull). It must be verified that the intersection of the plane with faces adjacent to v is a convex polygon. This can be done in $O(1)$ time with a number of processors that is equal to the degree of v. The total time to check that vertices are locally convex is $O(1)$ with $O(n)$ processors.

Next the result checker must verify that all of the points not said to be on the hull are really inside the boundary. For each point not on the hull, it finds a "proof" that it is indeed inside the boundary. This proof will consist of four points in the input set whose convex combination contains the nonhull point. The idea is to reduce the search for a proof to a planar point-location problem as follows: Choose the point p in the input set with minimum x-coordinate. Imagine a wall perpendicular to the x-axis at the maximum x-coordinate. Suppose that someone standing at p aimed and shot a blue paint gun at every point on the convex hull and along every edge. This would paint a triangulated planar graph on the wall. If the person standing at p then shot a red paint gun at every other point in the input set, there would be several red dots on the wall. If one is told which face of the planar graph on the wall a particular red dot landed in, then one would have the proof that is being sought, i.e., points reaching face (a, b, c) are exactly those points in the tetrahedron (p, a, b, c) (if any dot lands outside of the triangulation, then the point shot at must be outside of the convex hull). One must then just test that the point is really in the tetrahedron defined by (p, a, b, c). Determining (a, b, c) is simply a planar point-location problem. In [29] it is shown how to create the planar point-location data structure in $O(\log n)$ time with $n/\log n$ processors that supports point-location queries in $O(\log n)$ time.

Running Time. The checking and total time is $O(\log n)$ and the checking and total number of processors is $O(n)$.

6. Duality. When result checking an optimization problem, it is necessary to check that the solution is as good as is claimed, and that it is the best solution. Duality can sometimes be used to show the latter.

For example, to result check a program that does linear programming, the result checker need only check that the optimal solution is feasible, and to call the program again on the dual problem (again making sure that it is feasible) to check that the solution to the original problem is the same (and therefore optimal). If the program claims that there is no solution or that the solution is unbounded, this can be verified symbolically. This problem is P-complete, so no fast parallel algorithm is known for it. However, it can be result checked in logarithmic time with only two calls to the program using an obvious parallelization of the techniques in [18].

Another example is the following:

MAXIMUM MATCHING

Input: Graph $G = (V, E)$, where E is represented by an adjacency matrix.
Output: $k =$ the size of a maximum matching, and the edges in a maximum matching in G.

No deterministic NC algorithm is known for this problem, but it is known to be in RNC [21], [24].

Result-Checking Algorithm. The result checker first checks in parallel that no vertex is matched more than once and that the maximum matching is of size k. Then the algorithm in [19] is used to find a proof that there is no matching of size $\geq k$. This proof will be an odd set cover of size k. Karloff's algorithm calls a matching oracle on other problem instances. The result checker calls the matching program on these instances, and proceeds as if all of the answers are correct. If the output of his algorithm is an odd set cover of size $\neq k$, the result checker outputs "FAIL." Otherwise, the odd set cover of size k is verification that the maximum matching is of size k, and the result checker outputs "PASS."

Running Time. The result checking time is $O(d^{\mathrm{MIS}}(n))$ parallel steps and $O(p^{\mathrm{MIS}}(n))$ processors, where $d^{\mathrm{MIS}}(n)$ is the parallel depth and $p^{\mathrm{MIS}}(n)$ is the number of processors required to find a maximal independent set in an n node graph. The total running time is $O(d^{\mathrm{MIS}}(n) + D(n))$ with $O(n^3 \times N(n) + p^{\mathrm{MIS}}(n))$ processors.

7. Constant-Depth Reducible Functions. We can say something about the relationship among result-checking problems that are AC^0 equivalent.

PROPOSITION. *Let π_1, π_2 be two AC^0 equivalent computational problems. Then from any fast program result checker C_{π_1} for π_1, it is possible to construct a fast program result checker C_{π_2} for π_2.*

PROOF. Similar to Beigel's trick described in [7]. We outline the proof for decision problems, but the general proof is similar. The idea is to construct a program result checker for π_2 by transforming it to an instance of π_1 and result checking that instance. Since the oracle program still only solves π_2, in order to get an oracle for π_1 on x, we use the reverse transformation on x into an instance of π_2, and call the oracle for π_2 on it. Since the transformation and the reverse transformation can be computed in AC^0, the depth of the result checker for π_2 will be at most a constant times the depth of the result checker for π_1. Since π_1 and π_2 are AC^0 equivalent, the fastest parallel program for each is related by a constant factor. Therefore, if C_{π_1} is a fast program result checker, so is C_{π_2}. $\square$

Acknowledgments. We are especially grateful to Mike Luby for his many extremely helpful suggestions on this work and this manuscript, and to Dick Karp for pointing out that one can quickly check algorithms that can be solved by dynamic programming. We are grateful to Yossi Matias for his valuable comments and suggestions, as well as for pointing out the applications of the techniques to the problems of constructing breadth-first and depth-first search trees. We thank Mark Gross for pointing us to the result of [28] and we thank Klara Kedem and Raimund Seidel for helpful discussions about checking three-dimensional convex-hull programs. We are grateful to the referee for suggesting the third variant on the definition of quantifiablly different. We are also very grateful to Manuel Blum, Sampath Kannan, Russell Impagliazzo, Yishai Mansour, and Moni Naor for very helpful and interesting discussions.

References

[1] Adleman, L., Huang, M., and Kompella, K., Efficient Checkers for Number-Theoretic Computations, *Inform. and Comput.*, to appear.

[2] Aggarwal, A., and Anderson, R., A Random NC Algorithm for Depth First Search, *Combinatorica*, **8** (1988), 1–12.

[3] Aggarwal, A., Chazelle, B., Guibas, L., O'Dunlaing, C., and Yap, C., Parallel Computational Geometry, *Algorithmica*, **3** (1988), 293–327.

[4] Alon, N., Babai, L., and Itai, A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem, *J. Algorithms*, **7** (1986), 567–583.

[5] Amato, N., and Preparata, F., An NC^1 Parallel 3D Convex Hull Algorithm, *Proc. on Computational Geometry*, 1993, pp. 289–297.

[6] Beame, P., and Hastad, J., Optimal Bounds for Decision Problems on the CRCW PRAM, *J. Assoc. Comput. Mach.*, **36** (1989), 643–670.

[7] Blum, M., and Kannan, S., Program Correctness Checking ... and the Design of Programs that Check Their Work, *Proc. 22nd Symp. on Theory of Computing*, 1989, pp. 86–97.

[8] Blum, M., Luby, M., and Rubinfeld, R., Self-Testing/Correcting with Applications to Numerical Problems, *J. Comput. System Sci.*, **47**(3) (1993).

[9] Chandra, A., Fortune, S., and Lipton, R., Unbounded Fan-In Circuits and Associative Functions, *J. Comput. System Sci.*, **30** (1985), 222–234.

[10] Chandra, A., Stockmeyer, L., and Vishkin, U., Constant Depth Reducibility, *SIAM J. Comput.*, **13** (1984), 423–439.

[11] Cole, R., and Vishkin, U., Faster Optimal Parallel Prefix Sums and List Ranking, *Inform. and Comput.*, **70** (1986), 32–53.

[12] Furst, M., Saxe, J., and Sipser, M., Parity, Circuits and the Polynomial Time Hierarchy, *Math. Systems Theory*, **17** (1984), 13–28.

[13] Gazit, H., An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph, *Proc. Symp. on Foundations of Computer Science*, 1986, pp. 492–501.

[14] Goldberg, M., and Spencer, T., A New Parallel Algorithm for the Maximal Independent Set Problem, *Proc. Symp. on Foundations of Computer Science*, 1987.

[15] Gross, M., Irani, S., Rubinfeld, R., and Seidel, R., Personal communication.

[16] Leighton, F. T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.

[17] Luby, M., A Simple Parallel Algorithm for the Maximal Independent Set Problem, *SIAM J. Comput.*, **15** (1986), 1036–1053.

[18] Kannan, S., Program Result Checking with Applications, Ph.D. thesis, University of California, Berkeley, CA, 1990.

[19] Karloff, H., A Las Vegas RNC Algorithm for Maximum Matching, *Combinatorica*, **6** (1986), 387–392.

[20] Karp, R., and Ramachandran, V., A Survey of Parallel Algorithms for Shared-Memory Machines, Technical Report No. UCB/CSD 88/408, University of California, Berkeley, CA.

[21] Karp, R., Upfal, E., and Wigderson, A., Constructing a Perfect Matching is in Random NC, *Combinatorica*, **6** (1986), 35–48.

[22] Karp, R., Upfal, E., and Wigderson, A., The Complexity of Parallel Search, *J. Comput. System. Sci.*, **36** (1988), 225–253.

[23] Matias, Y., and Vishkin, U., Converting High Probability into Nearly-Constant Time—with Applications to Parallel Hashing, *Proc. Symp. on Theory of Computing*, 1991, pp. 307–316.

[24] Mulmuley, K., Vazirani, U., and Vazirani, V., Matching is as Easy as Matrix Inversion, *Combinatorica*, **7** (1987), 105–113.

[25] Preparata, F., and Shamos, M., *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[26] Schieber, B., and Vishkin, U., On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM J. Comput.*, **17**(6) (1988).

[27] Schönhage, A., and Strassen, V., Schnelle Multiplikation grosser Zahlen, *Computing*, **7**, 281–292.

[28] Spivak, M., *Differential Geometry*, Vol. 3.

[29] Tamassia, R., and Vitter, J. S., Optimal Parallel Algorithms for Transitive Closure and Point Location in Planar Structures, *Proc. ACM Symp. on Parallel Algorithms and Architectures*, 1989.