Achieving Optimal CRCW PRAM Fault-Tolerance

Alex A. Shvartsman

Brown University
Dept. of Computer Science
Providence, Rhode Island 02912

Technical Report No. CS-89-49
December 22, 1989

Achieving optimal CRCW PRAM fault-tolerance

Alex A. Shvartsman*

December 22, 1989

Abstract

As was shown in [KS 89], it is possible to combine efficiency and fault-tolerance in many PRAM algorithms in the presence of arbitrary dynamic fail-stop processor errors. Here we describe a technique for efficient and fault-tolerant simulation of *arbitrary* PRAM algorithms. Given a P-processor PRAM algorithm we simulate it efficiently and fault-tolerantly by a P'-processor CRCW PRAM algorithm, for $P' \leq P$. The simulation is optimal for $P' \leq P/\log^2 P$.

1 Introduction

An algorithmic approach to fault tolerance is defined in [KS 89]. This approach provides an effective way of transforming existing PRAM (parallel random access machine) algorithms [FW 78] so that they can be executed on fail-stop CRCW (concurrent read, concurrent write) PRAMs. Fail-stop processors are unreliable but can only fail by stopping. Such processors are used in [SS 83], as part of a methodology for developing fault-tolerant systems.

In the framework of [KS 89], when processor fail-stop errors occur, algorithms dynamically reconfigure their computation and successfully proceed to completion as long as there is at least one surviving processor. Failures are determined by an adversary that can stop processors at any time, except during a single bit PRAM write cycle. Most importantly, the efficiency of the original parallel algorithm is not significantly affected: for any pattern of fail-stop errors, the original *Parallel-time* $\times$ *Processors* performance is degraded, in the worst case, by a multiplicative factor polylogarithmic in the input size N using up to N processors. The efficiency of fault tolerant PRAM algorithms is measured in terms of *available processor steps*. This measure generalizes the standard *Parallel-time* $\times$ *Processors* product:

Definition 1.1 Consider an algorithm with P initial processors that terminates in parallel-time τ after completing its task on some input data I of size N and in the presence of fail-stop errors F. If $P_i(I,F) \leq P$ is the number of processors surviving at step i, then we define the available processor steps $S = S(N,P) = \max_{I,F}\{\sum_{i=1}^{\tau} P_i(I,F) : \text{data } I \text{ of size } N, \text{failure pattern } F \text{ for } P \text{ processors}\}$.

*Address: *Department of Computer Science, Brown University, PO Box 1910, Providence, RI 02912, USA* and *Digital Equipment Corporation, LKG2-2/T2, 550 King Street, Littleton, MA 01460, USA*. This work was supported in part by NSF grant IRI-8617344 and in part by ONR grant N00014-83-K-0146 ARPA order No. 4786.

In this work we develop a general technique for efficient simulation of *any* PRAM algorithm by a fail-stop CRCW PRAM. We formulate a *universal PRAM interpreter (UPI)*, then develop a fault-tolerant *UPI* that can execute any PRAM algorithm by storing the programs and processor registers in shared memory. A similar construction was independently developed in [KPS 89].

We conclude with an *optimal* execution of PRAM algorithms in the presence of arbitrary fail-stop errors. This is accomplished using our simulation together with a basic algorithm from [KS 89] that is optimal on inputs of size N when using P processors such that $P \leq N/\log^2 N$. The resulting fault-tolerant execution does not degrade the asymptotic efficiency of the source algorithm.

The fault-tolerant algorithms are executed on CRCW PRAMs. As was shown in [KS 89] the CREW (concurrent read, exclusive write) model is not sufficient. The CRCW algorithms that can be made fault-tolerant include the following models: WEAK (only zeros can be written concurrently), COMMON (concurrent writes of identical values are permitted), ARBITRARY (some single processor succeeds), PRIORITY (highest numbered processor succeeds), and STRONG (processor writing the largest value succeeds). [EG 88] and [KR 88] contain detailed surveys of PRAM models.

The problem we address is, at an intuitive level, one of controlling resource allocation. The resource controlled is all available PRAM processor steps, and it needs to be controlled because of the requirement to complete the computation regardless of faults. Note that unreliable PRAM processor steps must control all available PRAM processor steps. Distributed controllers have been developed for resource allocation such as the algorithms of [LGFG 86] (in a probabilistic setting) and of [AAPS 87] (in a deterministic setting). In each case the underlying models for computation, the resources controlled, and the actual algorithms and their analysis are very different.

The rest of this paper is structured as follows: in Section 2 we describe some background and give definitions, in Section 3 we show how to efficiently execute PRAM steps in the presence of fail-stop errors, and prove theorems on robust and optimal simulation of PRAM algorithms.

2 Background and definitions

It was shown in [KS 89] that an efficient fault-tolerant solution to a certain basic problem is fundamental in making efficient parallel algorithms fault-tolerant. This problem is defined as the *Write-All* problem: *given a N-processor PRAM and a zero-valued array of N elements, write value 1 into each array location.* In the absence of failures, this problem is solved in $O(1)$ time on a N-processor PRAM by a trivial and optimal "algorithm", since the best sequential time is $O(N)$. Simple fault-tolerant solutions to the *Write-All* problem exist (for example various processor clustering techniques), however all such solutions appear to be inefficient for many processor failure patterns. Thus a notion of *robustness* is introduced in [KS 89] to characterize algorithms that are both *efficient* and *fault-tolerant.* It is defined using the measure S (Definition 1.1):

Definition 2.1 Let $T(N)$ be the best sequential (RAM) time bound known for N-size instances of a problem. We say that a parallel (PRAM) algorithm for this problem is a *robust parallel algorithm* if: for any input I of size N and for any number of initial processors P ($1 \leq P \leq N$) and for any failure pattern F with at least one surviving processor, this algorithm completes its task and it has $S \leq c\,T(N)\log^{c'} N$, for fixed c, c'.

A robust solution to the *Write-All* problem is given in [KS 89] as the algorithm W. It is a four phase iterative algorithm. One of the phases is a *work phase*, where the work of the non-fault-tolerant algorithm is performed. Other phases estimate the progress made and dynamically allocate remaining processors to unfinished tasks. Using algorithm W, the following was shown:

Proposition 2.1 [KS 89] There is a robust parallel (COMMON or ARBITRARY) CRCW PRAM algorithm for the *Write-All* problem. This algorithm, on inputs of size N, has:

(1) $S = O(N \log N + P \log^2 N)$, when the initial number of processors P is between 1 and N.

(2) $S = O(N)$, when the initial number of processors P is between 1 and $N/\log^2 N$.

It was also shown in [KS 89] that any robust solution to the *Write-All* problem that uses $\Theta(N)$ initial processors will expend at least $\Omega(N \frac{\log N}{\log \log N})$ processor steps. Thus there are no optimal solutions to the N-processor *Write-All* problem. However, with an appropriate *slack* in the initial number of processors, it was possible to achieve optimal efficiency.

We say that a PRAM algorithm is *fault-tolerant* if it completes its task in the presence of arbitrary fail-stop errors. We say that a PRAM algorithm is simulated by a fault-tolerant PRAM algorithm when both algorithms exhibit identical input/output behavior. Such simulation is *robust* if it is *efficient*, and it is *optimal* if the efficiency of the simulating algorithm (measured by S) is within a constant factor of the efficiency of the simulated algorithm:

Definition 2.2 Let $\mathcal{M}$ be a simulation of a P-processor PRAM algorithm Q, whose parallel-time on inputs of size N is less than or equal to $\tau(N)$, by a fault-tolerant P'-processor algorithm Q'.

(1) $\mathcal{M}$ is *robust*, if Q' has $S = O(P \cdot \tau(N) \cdot \log^c N)$ for a constant c, and

(2) $\mathcal{M}$ is *optimal*, if Q' has $S = O(P \cdot \tau(N))$.

Note that a robust simulation of an algorithm is not the same as a robust algorithm. This is because the definition of robustness (2.1) is based on the best sequential algorithm, not an arbitrary parallel algorithm which may be inefficient. However a robust simulation of an efficient algorithm yields a robust algorithm.

Finally, it is relatively easy to construct robust simulations using a small number of processors, e.g. P' polylogarithmic in P. We will demonstrate that this is possible for P' in $1 \leq P' \leq P$, and optimally so for P' in $1 \leq P' \leq P/\log^2 P$.

3 Fault-tolerant execution of PRAM programs

PRAM programs are normally presented as Algol-like pseudo-code programs that can be compiled into assembly-like low level instructions using conventional techniques [W 79]. As is the case with sequential processors, the instructions are stored in memory, the address of an instruction to be executed next is stored in an *instruction counter* register, and in order to execute an instruction, it is fetched into an *instruction buffer*. When control structures such as **while-do** and **if-then-else** are used, they are compiled into assignments to instruction counters. Other processor local memory cells are stored in *registers*. Here we will use a formalization of the definition of PRAM given in [KR 88]. Intuitively, PRAM instructions consist of three synchronous cycles:

(1) *Read cycle*: a processor reads a value from a location in shared memory into local memory,

(2) *Compute cycle*: a processor performs a computation on local memory,

(3) *Write cycle*: a processor writes a value from local memory to a location in shared memory.

The formal definition is as follows:

```
forall processors PID=1..P parbegin
    shared SM[1..M]; --shared memory
    shared PROG[1..P,1..size]; --P programs of length size
    local IB --instruction buffer
    local r record IC, RR, WR, ... end --local registers
    r.IC := 1; --start at the first instruction
    while r.IC ≠ 0 do --while not HALT
        IB       := PROG[PID,r.IC]; --instruction fetch: IB now contains <R() C() W() J()>
        r.RR     := SM[R(r)]; --read cycle
        r        := C(r); --compute cycle
        SM[W(r)] := r.WR; --write cycle
        r.IC     := J(r); --next instruction
    od
parend
```

Figure 1: Universal PRAM Interpreter (*UPI*).

Definition 3.1 Q is a PRAM program on inputs of size N, if:

1. Q uses P processors with unique identifiers PID in the range 1 to P.
2. Each processor has an instruction buffer IB, and a set of internal registers collectively referred to as the record r. The number of registers per processor $|\mathbf{r}|$ is $O(\log^k N)$ for some constant k. The registers in r include an instruction counter IC, a read register RR and a write register WR used for reads/writes from/to shared memory.
3. Q uses M shared memory cells SM[1..M], with the first N cells containing the input. Shared memory cells and registers are capable of storing $O(\log(N+P))$ bits each.
4. Program instructions are stored in a shared array PROG[1..P,1..*size*], where *size* is a constant. Program for processor i is in PROG[i,1..*size*]), one instruction per array element.
5. Instructions consist of four encoded operations <R() C() W() J()> of size $O(\log N)$. After reading an instruction into IB, a processor interprets this code as follows:

```
RR       := SM[R(r)]; read cycle: read into RR the contents of shared memory location R(r),
r        := C(r);     compute cycle: assign to registers in r the result of computation C(r),
SM[W(r)] := WR;       write cycle: write the contents of WR to shared memory location W(r),
IC       := J(r);     compute next instruction address; IC = 0 is a halt.
```

□

R(), W() and J() are expressions involving registers, and C() is the code for individual processors' compute cycles. PRAM programs are executed by a *Universal PRAM Interpreter (UPI)* that accepts a PRAM program and its input as data and executes it. *UPI* is defined as the simple PRAM program in Figure 1 that formalizes the synchronous computation performed by a PRAM. In this section we develop a robust *UPI*.

Let us define $C_w(X,P)$ to be the cost (measured as S) of robustly solving the *Write-All* problem of size X, using P ($P \leq X$) processors. Such solution can be directly applied to zeroing an array of size X or copying X values from one array to another. Now the main lemma.

Lemma 3.1 Let Q be a P-processor COMMON (ARBITRARY) CRCW PRAM algorithm that uses M shared and $M_L = O(P\log^k N)$ (constant k) local memory on inputs of size N. Q can be simulated

```
01 forall processors PID=1..P parbegin
02     shared SM[0..1,1..M]; --two generations of shared memory
03     shared PROG[1..P,1..size]; --P programs of length size; one generation
04     shared IB[1..P]; --instruction buffers; one generation, local becomes shared
05     shared r[0..1,1..P] record IC, RB, WB, ... end --two generations of registers; now shared
06      --Action 0: initialize instruction counters
07     r[0,PID].IC := 1; --start at the first instruction
08     while r[0,PID].IC ≠ 0 do --while not HALT
09        --Action 1: tentative computation
10          IB[PID]              := PROG[PID,r[0,PID].IC]; --fetch <R() C() W() J()> into IB
11          r[1,PID]             := r[0,PID]; --copy registers to scratchpad
12          r[1,PID].RB          := SM[0,R(r[1,PID])]; --read cycle
13          r[1,PID]             := C(r[1,PID]); --compute cycle
14          SM[1,[W(r[1,PID])]   := r[1,PID].WB; --write cycle
15          r[1,PID].IC          := J(r[1,PID]); --next instruction
16        --Action 2: reconcile shared memory
17          SM[0,W(r[1,PID])]    := SM[1,W(r[1,PID])];
18        --Action 3: reconcile registers
19          r[0,PID]             := r[1,PID]
20     od
21 parend
```

Figure 2: Modified *UPI* using two generations of shared memory.

on a P'-processor ($P' \leq P$) fail-stop COMMON (ARBITRARY) CRCW PRAM using $O(M + M_L)$ shared memory, at the cost of $S = O(C_w(P, P') \cdot \log^k N)$ per parallel PRAM step.

Proof: The desired result is achieved by constructing a robust version of *UPI* of Figure 1. Figure 2 illustrates an intermediate step of the construction, and Figure 3 is the final pseudo-code for the robust *UPI*. The proof consists of the following six steps: (1) local memory is stored in shared memory, (2) shared memory is split into two generations, "current" and "future", (3) PRAM step computations are changed to use current memory as input and use future as output and as a computation scratchpad, (4) current and future memories are reconciled to produce new current memory, (5) instruction counters are examined to detect termination, and finally, (6) each of the modified groups of actions is placed in the work phase of algorithm W of [KS 89]. Now the details.

First, we have to store processor local memory in shared memory, since after processors fail, the local memory is lost. We observe that M_L local memory cells can be stored in shared memory without affecting program semantics. To do this, registers are subscripted by PID (lines 04-05, all lines refer to Figure 2).

We then separate all shared memory and registers into two generations: current (0) and future (1) (lines 02 and 05). All memory references are now made using the generation subscript 0 or 1 (lines 10-19). This does not affect the asymptotic memory requirement of size $O(M + M_L)$. This is done to assure that the memory that can be accessed by processors that have not yet completed a particular action (due to failures) is not changed. This allows the PRAM instructions to be restarted when active processors are re-allocated.

Next we group statements into four actions to compute future memory values and to reconcile current and future memories. Action 0 is the initialization of ICs. The contents of the while loop of Figure 1 are groupped into 3 actions: Action 1 (lines 09-15) performs the tentative PRAM step computation using current memory as input, and future memory as output and as a scratchpad; Action 2 (lines 16-17) reconciles shared memory; and Action 2 (lines 18-19) reconciles processor

forall processors PID=1..P' **parbegin**
shared PROG[...], SM[...], IB[...], r[...], *H1*, *H2*, *H3*, HALT **initial** *false*;
a: Use W with *H1* for Action 0 (initialize ICs to 1); Use W with *H3* to clear *H1*;
while not HALT **do** ——*while not all processors halted*
b: Use W with *H1* for Action 1 (tentative PRAM step); Use W with *H2* to clear *H1* and *H3*;
c: Use W with *H1* for Action 2 (reconcile memory); Use W with *H3* to clear *H1* and *H2*;
d: Use W with *H1* for Action 3 (reconcile registers); Use W with *H2* to clear *H1* and *H3*;
e: Use W with *H1* to compute HALT=*false* iff ∃PID s.t. IC≠0; Use W with *H3* to clear *H1* and *H2*;
od
parend

Figure 3: Pseudo-code for a robust *UPI*.

registers.

Now algorithm W is used with each of the four actions as work phases, see Figure 3, steps a, b, c and d. This assures that all actions of a given phase are performed before any of the actions of the next phase are attempted.

Algorithm W utilizes workspace memory of size $O(P)$ on inputs of size P using P' processors ($P' \leq P$). This memory initially contains zeroes. Consecutive use of the algorithm W in a single algorithm requires that this workspace is zeroed. This is accomplished by utilizing three interchangeable workspaces, call them *H1*, *H2* and *H3*. *H1* is used in the actual algorithm. *H2* is used in zeroing *H1* and *H3* using algorithm W. *H3* is saved for the next use of the zeroing cleanup step. *H2* and *H3* then alternate. This simple, but modular cleanup technique affects neither the overall asymptotic memory usage, nor the asymptotic efficiency of robust algorithms when the cleanup stages are interleaved with the computation stages (Figure 3).

Lastly, we need to check for the algorithm termination by verifying that all processors halted. This is done by computing shared HALT to be *true*, iff for all PIDs, IC=0. This can be accomplished in unit time on a CRCW PRAM in the absence of failures. Here we compute HALT as follows: it is initialized to *true*, then using the *Write-All* technique, processors examine the simulated ICs and execute HALT:=*false* for each IC≠0 (Figure 3, step e).

In this simulation, the tentative PRAM step computations may occur asynchronously due to failures, however since no processor reads or writes registers of other processors, and since the program PROG[...] is either COMMON or ARBITRARY CRCW, it does not matter in what order the shared memory is written by the simulated processors. Therefore the simulated PRAM steps will write values to shared memory in a way that is consistent with both the COMMON and ARBITRARY models.

To analyze the complexity of the resulting fault-tolerant computation, we first examine the use of registers. In actions 1 and 3 of Figure 2, each PRAM processor may need to compute the values of, and copy the shared memory that represents a processor's registers as the result of interpreting the computation C(). This must be done sequentially by each processor, thus incurring a polylogarithmic in N multiplicative overhead. However this overhead still falls within the definition of robustness (2.1).

The complexity of applying this technique to a single PRAM step is bounded by the complexity of solving the *Write-All* problem in steps a, c and d of Figure 3, plus the complexity of robustly writing and copying $M_L = O(P \log^k N)$ shared memory in steps b and d. To copy M_L memory, we apply the *Write-All* technique M_L/P times at the cost of $C_w(P, P')$ per application. Therefore, the

total cost per single PRAM step is $S = O[C_w(P,P') + (M_L/P)C_w(P,P')] = O[C_w(P,P') + \log^k N \cdot C_w(P,P')] = O(C_w(P,P') \cdot \log^k N)$. □

A construction for PRAM simulation, similar to the one used in Lemma 3.1 was independently developed in [KPS 89] using the *Write-All* technique. In [KPS 89] it is also observed that since P processors can only read P and write P shared memory cells in a single PRAM step, the overhead in shared memory need only be $O(P)$ when processors have no local memory. The results that follow do not take advantage of this optimization.

Theorem 3.1 Any P-processor PRAM (EREW, CREW, and WEAK, COMMON, ARBITRARY, PRIORITY or STRONG CRCW) algorithm can be robustly simulated on a fail-stop P'-processor COMMON or ARBITRARY CRCW PRAM, when $P' \leq P$.

Proof: For COMMON or ARBITRARY CRCW PRAMs, Proposition 2.1(1) establishes $C_w(P,P) = O(P \log^2 P)$. Then the proof follows from Lemma 3.1 and Definition 2.2(1).

Correct EREW (exclusive read, exclusive write), CREW and WEAK CRCW PRAM programs can be directly executed on a CRCW PRAM without changing the program semantics. Therefore the technique of Lemma 3.1 can also be used with CREW and EREW algorithms that are to be executed on a CRCW PRAM. Thus the result holds for EREW, CREW, and WEAK CRCW models.

To extend the simulation to stronger CRCW models such as PRIORITY and STRONG, we use an efficient algorithm transformation technique that preserves algorithms' efficiency to within a logarithmic in the number of processors factor as shown in [EG 88] (attributed to folklore): "A parallel computation that can be performed in time τ on a P-processor STRONG CRCW PRAM, can also be performed in time $\tau \log P$ using P EREW processors". Therefore we first preprocess PRIORITY and STRONG PRAM algorithms, and then robustly simulate the transformed algorithms on fail-stop COMMON or ARBITRARY CRCW PRAMs. □

To achieve optimal simulation, we are going to use the algorithm W within its range of optimality:

Theorem 3.2 Any P-processor PRAM algorithm that uses constant local memory per processor can be optimally simulated on a fail-stop P'-processor CRCW PRAM, when $P' \leq P/\log^2 P$. EREW, CREW, and WEAK and COMMON CRCW PRAM algorithms are simulated on fail-stop COMMON CRCW PRAMs; ARBITRARY, PRIORITY and STRONG CRCW PRAMs are simulated on fail-stop CRCW PRAMs of the same type.

Proof: For the optimality result we use Proposition 2.1(2) that establishes $C_w(P, P/\log^2 P) = O(P)$. We also eliminate the overhead of copying local memories by allowing constant local memory per processor. In this case the cost of copying local memory is absorbed by the cost of optimal simulation of PRAM instructions by using Lemma 3.1 with $k = 0$ (constant local memory). This proves the result for for EREW, CREW, and WEAK, COMMON and ARBITRARY CRCW PRAMs.

To prove the result for PRIORITY PRAM, we need to use a property of algorithm W [KS 89] that we call *processor allocation monotonicity*. This property states that processors are allocated to PRAM steps of simulated processors in such a way that if processors with PIDs p_1 and p_2 (w.l.o.g. let $p_1 < p_2$) are simulated respectively by processors with PIDs p'_1 and p'_2, then $p'_1 < p'_2$. Having this property assures that if two or more PRIORITY processors write concurrently, then the processor that simulates the highest numbered processor will succeed. It has to be also assured that during a single PRAM step simulation, lower numbered processors do not overwrite cells written

asynchronously by higher numbered processors. This is done using auxiliary storage as in the the transformation in [EG 88]: before writing, processors first write the PID of the simulated processor and then the data, but only if the previously written PID is lower.

To show that algorithm W has the *processor allocation monotonicity*, we need to examine the static bottom up traversal procedure S_BU() in Appendix A.2 and dynamic top down traversal procedure D_TD() in Appendix A.4 of [KS 89]. In procedure S_BU(), a surviving processor is enumerated by adding one to the overestimate of surviving processors with PIDs smaller than the processor's PID. Thus processors are monotonically enumerated (larger PIDs are given larger processor numbers). In procedure D_TD(), the enumerated processors are assigned in a divided-and-conquer strategy according to a binary tree hierarchy. Lower numbered processors are assigned to the subtrees that contain lower numbered work elements and higher numbered processors are assigned to the subtrees that contain higher numbered work elements. In the context of PRAM step simulation, the work elements are associated with the PIDs of simulated processors. Therefore the processor allocation is monotonic.

For the simpler case of STRONG PRAM, the concurrent writes are properly handled by first zeroing the future memory cells to which processors will write, and then performing writes only if the value in the cell is smaller than the value to be written (w.l.o.g. use unsigned integers). This assures the correctness of asynchronous writes. Synchronous writes are properly handled by the STRONG PRAM itself since regardless of the processor used, the writes of larger values will succeed. □

Acknowledgements: We thank Paris Kanellakis and Zvi Kedem for helpful discussions.

4 References

[**AAPS 87**] Y. Afek, B. Awerbuch, S. Plotkin, M. Saks, "Local management of a global resource in a communication network", *Proc. of the 28th IEEE FOCS*, pp. 347-357, 1987.

[**EG 89**] D. Eppstein and Z. Galil, "Parallel Techniques for Combinatorial Computation", *Annual Computer Science Review*, 3:233-83, 1988.

[**FW 78**] S. Fortune and J. Wyllie, "Parallelism in random access machines", *Proc. 10th ACM STOC*, pp. 114-118, 1978.

[**KS 89**] P. C. Kanellakis and A. A. Shvartsman, "Efficient parallel algorithms can be made robust", Brown Univ. Tech. Report CS-89-35 (submitted to *Distributed Computing*); prel. version appears in *Proc. of the 8th ACM PODC*, pp. 211-222, 1989.

[**KR 88**] R. M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines", to appear in the *Handbook of Theoretical Computer Science*, to be published by North-Holland.

[**KPS 89**] Z. M. Kedem, K. V. Palem, P. Spirakis, "Efficient Robust Parallel Computations", unpublished manuscript, 1989.

[**LGFG 86**] N.A. Lynch, N.D. Griffeth, M.J. Fischer, L.J. Guibas, "Probabilistic analysis of a network resource allocation algorithm", *Information and Control*, vol. 68, pp. 47-85, 1986.

[**SS 83**] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerant computing systems", *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222-238, 1983.

[**W 79**] J. C. Wyllie, *The Complexity of Parallel Computation*, Ph.D. Thesis, Cornell University, TR 79-387, 1979.