THE COMMUNICATIONS COMPLEXITY HIERARCHY IN DISTRIBUTED COMPUTING

J. B. Sidney+ and J. Urrutia*

1. INTRODUCTION

Since the pioneering research of Cook [1] and Karp [3] in computational complexity, an enormous amount of research has been directed toward establishing the position of problems in the complexity hierarchy: polynomial, weakly NP-complete, strongly NP-complete, etc. In rough terms, the computational complexity of a problem is an asymptotic measure of the "amount of time" needed to solve all instances of a problem using a finite state Turing machine with an infinitely long tape. The complexity hierarchy as currently understood depends for its existence on the assumption that $P \neq NP$, i.e., that there are problems solvable in exponential time but not in polynomial time. Since the computational complexity for a problem π on any real digital computing system is bounded by a polynomial transformation of the Turing machine complexity, it follows that the Turing machine complexity hierarchy is equally valid for the RAM model of computation.

With the advent of distributed computer systems, a new emphasis must be placed upon establishing the separate complexities of local processing activities (e.g., activities within a single processor) and of communication activities (e.g., information transfers between processors). More specifically, since communication activities tend to be slower and less reliable than local processing activities, algorithm designers should consider the trade-off between the two.

In this paper, we show (section 2), that NP-hard problems are intrisically "hard" also with respect to communication activities, while (section 3) "easy" (P) problems are intrinsically "easy" with respect to communication activities.

+Faculty of Administration, University of Ottawa, Ottawa, Ontario, Canada

*Department of Computer Science, University of Ottawa, Ottawa, Ontario, Canada.

This research was supported under a Natural Sciences and Engineering Research Council (Canada) research grant.

2. COMMUNICATION COMPLEXITY FOR "HARD" PROBLEMS

Consider a computer system C=(**P**,**S**) consisting of a finite state processor P and a countably infinite periferal storage devise **S.** Processor P has a storage capacity of $\leq$ m bits, where m includes system occupied space, register space, primary (rapid access) memory, etc. Define the state of C to be a (countable) vector □ = ($□_p$, $□_s$), the concatenation of the state of $□_p$ of P and the state $□_s$ of S. Since P contains at most m bits, there are at most 2^m possible values $□_p$ could assume.

Suppose Π is an NP-hard problem, solvable by algorithm A on C under a "reasonable" encoding scheme E, where algorithm A cannot cycle (As discussed in Garey and Johnson [2], no definition of "reasonable" encoding scheme is known, although this notion is essential to complexity results). For any problem instance π□Π, define $e(\pi)$ to be the number of bits required to encode Π under E. We shall take as our measure of complexity the number of state transitions taking place during execution of algorithm A, where a state transition may occur at any integer time $t > 0$ and involves a change in at most a constant w bits. For example, w could be the world length in the system C. A communication between P and S is defined to be a state transition involving a change in $□_s$.

Let $h(n) = \text{Max}\ \{\text{number of state transitions of C to solve } \pi \mid e(\pi) \leq n\}$ and let $g(n) = \text{Max}\ \{\text{number of communications between p and s to solve } \pi \mid e(\pi) \leq n\}$.

Theorem 1: Assume P $\neq$ NP and Π □ P. Then there exist no constants a and b such that $g(n) \leq a\, n^b$ for all $n > 0$.

Proof: Given a and b, set $\underline{a} = 2^m a$. Since Π □ P, there must be some value of n such that $h(n) > \underline{a}\, n^b$. Let n_o be such that $h(n_o) > \underline{a}\ n_o^b$. Between any two successive communications between P and S there can be at most 2^m state transitions involving changes to only $□_p$ (otherwise there would be cycling). Therefore $g(n_o)>h(n_o)/2^m=a\, n_o^b$, from which the result follows. []

3. COMMUNICATION COMPLEXITY FOR 'EASY' PROBLEMS

The results in this section require for their proof a more formal approach based directly on the Turing machine computational model. Our presentation below is based directly on the discussion and notation in Garey and Johnson [2].

There are five procedures which the DTM performs at each iteration; q is the current state at the start of an iteration, and □ is a finite set of tape symbols including q_o (start), q_Y and q_N (yes and no terminate states), and a blank.

READ: read contents □of tape square being scanned

TRANSFORM: compute □(q, □) = (q', □,□) and set q   q'
HALT: if current state = q_Y or q_N, halt execution
WRITE: replace □ with □ on tape square being scanned
POINTER: translate read-write head (pointer) by □.

If a problem ∏ belongs to P under a "reasonable" encoding scheme E, then by definition there is a polynomial time DTM program M that "solves" ∏ under the encoding scheme E. In other words, the number of steps or iterations $G(\pi)$ to solve any instance π □∏ is bounded by a polynomial function $f(e(\pi))$, i.e. $G(\pi) \leq f(e(\pi))$ for all π□∏. Subject to certain reasonably non-restrictive assumptions, we shall show how M can be realized on a distributed system with the number of required communication activities (to be defined) bounded by a polynomial in $e(\pi)$.

In order to accomplish this, we define below a model of distributed computer and prove the desired result for this system model. We belive the model is flexible enough to be adapted to deal with most real systems.

Let the computer system $C=(S_o,\mathcal{S})$ consist of a finite state processor S_o and a countable set $\mathcal{S}$ of periferal storage devices S_j ($j= \pm1, \pm2,...$). S_j ($j\neq0$) has a storage capacity of $m_j+1\geq m$ words ($m\geq1$ an integer), each word having at least Max{ □og$_2(m_j+1)$□, □} where □ is the number of bits required to store the bit-wise longest element of □. S_o will have a reserved storage area consisting of one word whose content is denoted by $S_o(1)$, and of bit lenght at least □. The words in S_j ($j\neq0$) are addressed by the numbers from 0 to m_j; and $S_j(i)$ denotes the contents of the word in S_j whose address is i. Note that word length at site $j\neq0$ is sufficiently large to accommodate the binary expansion of m_j . Included in S_j is a small processor P_j whose functions are described later.

The sites S_j ($j = 0, \pm1, \pm 2, ...$) are serially linked by two-way communication lines, so that for all j, S_j has the capability to communicate directly (send to and receive messages from) S_{j-1} and S_{j+1}. No other communication lines exist between the sites.

An equivalence between the DTM tape positions and the positions in the sites S_j is defined as follows, where the empty sum by definition =0.

- for $j \geq1$, S_j (h) corresponds to DTM tape position $\;_{i=1,j-1} m_i + h$ for (($1\leq h \leq m_j$), $1\leq j$),
- for $j \leq -1$, S_j(h) corresponds to DTM tape position $\sum_{i=1,j-1} m_{-i} - h$ for (($1\leq h \leq m_j$, $j \leq 1$)
- $S_0(1)$ corresponds to tape position 0. The first word S_j (0) ($j\neq0$) is a pointer;

the set of pointers provides the distributed system with the position of the "square" currently being scanned, as described later. For any DTM tape square t, define $\underline{S}(t) = S_j$ where S_j contains the image of t, and let $\underline{T}(t)$ = address in $\underline{S}(t)$ of the image of tape square t. The vector $(\underline{S}(t), \underline{T}(t))$ is called the position of tape square t.

The processor S_o plays the role of the finite state control, and "contains" the transition function □ and the current state indicator q. S_o contains sufficient storage to execute the tasks assigned to it; e.g., calculation of □, message transmission and reception.

A basic communication activity (BCA) is the sending from S_j and reception by S_{j-1} or S_{j+1} of a message of the form (a,b) where a □□ □{0} and b□{-1,0,1} (without loss of generality, assume 0 is not in □). It is assumed that all sites S_j have the capability of sending and receiving messages as described.

The execution of algorithm M on C is straightforward. Initially, the system memory is assumed to have contents as follows:

(i) the DTM input string x is stored in positions $(\underline{S}(1), \underline{T}(1)) = (1,1)$ to $(S_h, T(|x|) = (\underline{S}(|x|), \underline{T}(|x|)$.

(ii) $S_j(0) = 0, \quad j \neq 0$

(iii) $S_h(i)$ = blank; $\underline{T}(|x|) < i \leq m_h$

(iv) $S_j(i)$ = blank, $((i \leq i \leq m_i), j$ □ $\{1, ..., h\})$

(v) Current state $q = q_o$

The execution of algorithm M on C is accomplished by exploiting the correspondence between the memory storage of the processors and tape positions; any time an operation (read, write, transform) must be executed by M at tape position t a control token (as well as the specification of the function to be applied) will be sent from P_o to the processor P_j having the storage device associated with tape position t; P_j will then execute the operation and send the control token and the result of the operation back to P_o which will determine the next step to be performed. In a similar manner, a move operation can be executed.

A formal description of the simulation of algorithm M on C can be found in the appendix, where Table 1 describes the five procedures READ, TRANSFORM, HALT, WRITE, and POINTER, and Table 2 specifies precisely the messages transferred from site to site. Column 1 of Table 1 gives an upper bound on the number of BCA's required for eachj procedure, where n=|x|, f(n) is the DTM complexity function, and the precise description of the system communication rules is given in Table 2. k= □(f(n)+1/m□ is a constant of the problem.

Theorem 2: Let M be a DTM algorithm that requires at most f(n) iterations to solve any problem whose encoding x has length $\leq$ n, where x is the "reasonable" encoding of $p \in P$. Then the distributed version of algorithms M on C = (S_0, S) requires at most $1+\lceil f(n)+1/m \rceil f(n)$ basic communication activities.

Proof: Clearly, f(n) is an upper bound on the number of iterations, and hence f(n) +1 is an upper bound on the number of positions in memory required. Thus, $\lceil (f(n)+1) / m \rceil$ is an upper bound on the number of sites S_j ($j \neq 0$) accessed during execution.
Examination of Table I in the appendix makes it evident that the sequence of procedures READ, TRANSFORM, HALT, WRITE, POINTER will require at most one message in each direction between each two adjacent sites during any complete iteration. Thus, an upper bound of $1 + f(n)\lceil (f(n)+1) / m \rceil$ BCA's is required to execute M on C, where 1 corresponds to the initial message from S_0 to S_1. □

We have explored the relationship between the serial computational complexity of a problem Π and the communications complexity of solving Π on a distributed system. In broad terms, the communication complexity hierarchy is "inherited" from the serial complexity, that is, a problem which is NP hard with respect to serial complexity has been shown to be NP hard with respect to the communication complexity and any algorithm with polynomial serial complexity can be realized in a distributed system with polynomial communication complexity. The latter result is based upon a particular model of a distributed system.

REFERENCES

[1] S. Cook, "The complexity of theorem-proving procedures", *Proc. 3rd ACM Symp. on Theory of Computing*, 1970, 151-158.

[2] M.R. Garey, D.S. Johnson, "Computers and Intractability: a Guide to the Theory of NP-Completeness", *Freeman*, San Francisco.

[3] R.M. Karp, "Reducibility among combinatorial problems", in "Complexity of Computer Computations", *Pergamon Press*, New York, 1972.

APPENDIX

TABLE 1

PROCEDURE	UPPER BOUND ON NUMBER OF BCA'S PER ITERATION	HOW PROCEDURE IS ACCOMPLISHED (Note: "Messages" from S_o to S_o are executed internally in S_0)
READ	K	Set S_j= <u>S</u>(t). S_j sends the message □=S_j(□(t)) □□ S_o.
TRANSFORM	0	Internally, S_0 computes □(q, □) = (q', □', □), where □ =q and □ is the contents of the position (S(t), T(t)) currently being scanned. S_0 sets q q'
HALT	0	Internally, S_0 determines if q = either q_Y or q_N. If yes, S_0 terminates execution.
WRITE	K	S_0 sends the message (□', □) to <u>S</u>(t). <u>S</u>(t) writes □ at position (<u>S</u>(t), <u>T</u>(t)).
POINTER		Let S_j = <u>S</u>(t) and S_k = <u>S</u>(t+□). Necessarily, k = (j-1), j, or (j+1)
	Case 1:0	Case 1: j=k≠0 □□ □□□□ □□ (0) S_j (0) + □
	Case 2:1	Case 2: j ≠ 0, k≠0, j≠k S_j sets S_j (0) 0. There are four cases:

	j < 0	j > 0
k = j-1	S_j sends message to S_{j-1} to set S_{j-1}(0) = 1 and commence READ	S_j sends message to S_{j-1} to set S_{j-1}(0) =m_{j-1} and commence READ
k = j+1	S_j sends message to S_{j-1} to set S_{j-1}(0) =m_{j+1} and commence READ	S_j sends message to S_{j-1} to set S_{j-1}(0) =1 and commence READ

	Case 3: 1=2/2 (2 BCA's. But	Case 3: (j=-1,k=0) or (j=+l,k=0). This case is special, and includes <u>in addition to</u> the initial POINTER procedure, a complete iteration. S_j sets S_j(0) 1 and sends message to S_0 that current

POINTER	position being scanned is $S_0(1)$.
procedure	<u>Internally</u>, S_0 executes READ, TRANSFORM, HALT
is executed	and WRITE. Let TRANSFORM result in (q', □, □),
for two	and let S_h = <u>S</u>(□).
iterations.)	S_0 sends message to S_h to set $S_h(0)=1$ and commence READ

TABLE 2 - MESSAGE DEFINITION

Receiving site	Transmitting site	Message	Response	Associated Procedures (Table 1)
S_0	S_1	(□1)	(*i*) S_0 computes □(q,□)=(q',□,□)	TRANSFORM
			(*ii*) q q'	"
			(*iii*) If $q=q_y$ or q_N, stop.	HALT
			(*iv*) Message (□,□) sent to S_1	WRITE, POINTER
	S_1	(0,0)	Note: this occurs when pointer goes from S_1 to S_0	
			(*0*) S_0 sets □= $S_0(1)$	POINTER, READ
			(*i*) as above	⊇ as above
			(*ii*) as above	□ as above
			(*iii*) as above	□ as above
			(*iv*) S_0 sets $S_0(1)$ □	WRITE
			(*v*) Message (0,0) sent to <u>S</u>(□)	POINTER
	S_{-1}	(□1)	(*i*) as above	⊇ as above
			(*ii*) as above	□ as above
			(*iii*) as above	□ as above
			(*iv*) Message (□,□) sent to S_{-1}	WRITE, POINTER
	S_{-1}	(0,0)	Note: this occurs when pointer goes from S_{-1} to S_0	
			(*0*) S_0 sets □= $S_0(1)$	POINTER, READ
			(*i*) as above	⊇ as above
			(*ii*) as above	□ as above
			(*iii*) as above	□ as above
			(*iv*) S_0 sets $S_0(1)$ □	WRITE
			(*v*) Message (0,0) sent to S(□)	POINTER
$S_j(j\geq1)$	S_{j-1}	(□,□)	(*i*) If $S_j(0)=0$ send (□,□) to S_{j+1}	WRITE, POINTER
			(*ii*) If $S_j(0)\neq0$ then S_j(<u>T</u>(t)) □ and (a), (b), or (c) is executed as required:	WRITE

			(a) If $\underline{S}$(t+□)=S_j	
			Set S_j(0)= S_j(0)+□	POINTER
			Set □= S_j(S_j(0)) and send	
			message (□,1) to S_{j-1}	READ
			(b) If $\underline{S}$(t+□)= S_{j+1}	
			Set S_j(0)=0	POINTER
			Send message (0,0) to S_{j+1}	POINTER
			(c) If $\underline{S}$(t+□)=S_{j-1}	
			Set S_j(0)=0	POINTER
			Send message (0,0) to S_{j-1}	POINTER
	S_{j-1}	(0,0)	(*i*) Set S_j(0)=1	POINTER
			(*ii*) Set □= S_j(1)	READ
			(*iii*) Send message (□,1) to S_{j-1}	READ
	S_{j+1}	(□,1)	Send message (□,1) to S_{j-1}	READ
	S_{j+1}	(0,0)	(*i*) Set S_j(0)=m_j	POINTER
			(*ii*) Set □=S_j(m_j)	READ
			(*iii*) Send message (□,1) to S_{j-1}	READ
$S_{(-j)}$(j≥1)	....		Reverse the sign of all subscripts in the table for S_j (j≥1) to get the corresponding entries for $S_{(-j)}$(j≥1).	....