ware and fault-tolerant computing. In 1979 he joined the Information Sciences Institute of the University of Southern California, Marina del Rey, where he has carried out research into hardware descriptive languages, emulation of machine descriptions, and microcode verification. He is currently supervising a project at USC-ISI, investigating strategies for the testing and debugging of formal semantic definitions of programming languages (in particular, Ada) by transforming such definitions into executable program objects. His current research interests include fault-tolerant computing, reliability modeling, formal semantics of programming languages, design-automation of digital hardware, and the design of hardware descriptive languages.

Dr. Kini is a member of the IEEE Computer Society and Phi Kappa Phi.

Daniel P. Siewiorek (S'67–M'72–SM'79–F'81), for a photograph and biography, see page 671 of the July 1982 issue of this TRANSACTIONS.

Distributed Reconfiguration Strategies for Fault-Tolerant Multiprocessor Systems

EDMUND M. CLARKE, MEMBER, IEEE, AND CHRISTOS N. NIKOLAOU

***Abstract*—In this paper, we investigate strategies for dynamically reconfiguring shared memory multiprocessor systems that are subject to common memory faults and unpredictable processor deaths. These strategies aim at determining a *communication page*, i.e., a page of common memory that can be used by a group of processors for storing crucial common resources such as global locks for synchronization and global data structures for voting algorithms. To ensure system reliability, the reconfiguration strategies must be *distributed* so that each processor *independently* arrives at exactly the same choice. This type of reconfiguration strategy is currently used in the STAGE operating system on the PLURIBUS multiprocessor [5]. We analyze the weak points of the PLURIBUS algorithm and examine alternative strategies satisfying optimization criteria such as maximization of the number of processors and the number of common memory pages in the reconfigured system. We also present a general distributed algorithm which enables the processors in such a system to exchange the local information that is needed to reach a consensus on system reconfiguration.**

***Index Terms*—Communication page, fault-tolerence, multiprocessor systems, reconfiguration strategies.**

I. INTRODUCTION

THE use of multiprocessor systems in real-time applications, such as network or aircraft controllers, considerably increases the reliability requirements of such systems. The presence of multiple resources can potentially enable a multiprocessor system to continue functioning despite nonfatal hardware errors. To achieve this goal, however, it is necessary to develop adequate software to detect failures and, if possible, reconfigure the system into a reliable subsystem.

In this paper, we consider shared memory multiprocessors where memory is partitioned into *common memory*, accessible by all processors, and *local memory* associated with a particular processor and only accessed by that processor. We focus our attention on two important classes of failures: memory failures and processor deaths. Furthermore, we assume that the memory failures never appear as arbitrary alterations of memory contents, but rather as unaccessibility of parts of the shared memory by some or all processors. Likewise, "death" is the only symptom of malfunction that we assume for processors. Failure of memory local to a particular processor will appear as slowdown or death of the processor and will not be considered.

In order to make such a multiprocessor system fault tolerant it is necessary to develop strategies which, after the occurrence of one or more memory failures or processor deaths, will permit the processors to cooperate and agree on a reconfiguration of the system. The resulting subsystem should be one, where all participating processors can access all parts (pages) of participating common memory, and therefore share common resources.

An important first step in such a strategy is the determination of a *communication page*, a page of common memory that can be used by a group of processors as a message area while they attempt to arrive at a consensus on the appropriate reconfiguration of the system through various voting procedures. Determination of the communication page is complicated by the fact that no processor can assume the existence of global data structures or global locks for synchronization until this page has been selected. This problem is solved by the STAGE operating system on the PLURIBUS multiprocessor [5] by means of a *distributed algorithm* which forces each processor in the reconfigured system to independently arrive

Manuscript received October 6, 1981; revised January 21, 1982 and February 16, 1982. This work was supported by the National Science Foundation under Grant MCS-7908365 and by the U.S. Navy, Naval Electronic Systems Command, under Contract N00039-78-G-0020.

E. M. Clarke is with the Center for Research in Computing Technology, Harvard University, Cambridge, MA 02138.

C. N. Nikolaou is with IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

at the same choice of communication page and shut down those processors which cannot see this page.

Since a given page of common memory will, in general, only be visible to a subset of the processors, the choice of a communication page can seriously affect the size of the reconfigured system. Thus, it is important to develop criteria for measuring the performance of alternative approaches for selecting a communication page. The criteria that we have selected deal with optimization of the two main system resources, namely processors and pages of common memory:

1) maximization of the number of participating processors in the resulting subsystem,

2) maximization of the number of pages of common memory used by the participating processors,

3) maximization of the value of an arbitrary performance function of both the processors and the pages.

In what follows, we analyze the algorithm used by PLURIBUS for establishing a communication page. This algorithm yields configurations with maximum number of common pages if single faults are assumed. We show, however, that it does not produce an optimum choice according to either of the first two optimization criteria, in the case of multiple faults. Next, we present a general algorithm which enables processors to exchange private information. We show how this algorithm can be used to develop reconfiguration strategies satisfying the first two criteria. Finally, we prove that the decision problem associated with the satisfaction of the third criterion is NP-complete, and develop and analyze probabilistic algorithms which yield system configurations approximating the maximum of special performance functions.

II. The Model

Our multiprocessor model is loosely based on the PLURIBUS system [5] developed by Bolt Beranek and Newman as a reliable, high-speed Interface Message Processor (IMP) for the ARPANET. We expect, however, that our results are applicable to many other shared memory multiprocessor systems. Fig. 1 gives an overview of the architecture of the PLURIBUS system. The system is composed of the three different types of buses (processor, memory, and I/O) joined together by special bus couplers which permit units on one bus to access units on another. Each processor bus contains some number of processors together with their associated local memory. The memory buses contain the pages of common memory that are accessible to all processors; thus, each processor bus is connected to every memory bus. The I/O buses are used to provide an interface with various I/O devices and will not be considered in our model. We distinguish three possible types of failures for such a configuration.

1) An individual processor can die or a processor bus can fail.

2) Several pages of common memory or perhaps an entire memory bus can fail.

3) A bus coupler connecting a processor bus and a memory bus can fail.

In each case the effect of the failure is to change the set of common memory pages that can be accessed by some set of processors. We assume that each failure type can be detected by some combination of hardware and software. A failed memory bus, for example, will cause a hardware trap when a processor attempts to access some word in a memory page on the bus.

Consider a multiprocessor system where $P = \{p_1, \cdots, p_p\}$ is the set of processors with cardinality p, and $M = \{m_1, \cdots, m_m\}$ is the set of common pages with cardinality m. The *system-map* or *P-M graph* is the binary relation

$$S = \{(p_i, m_j)/p_i \text{ accesses page } m_j\}$$

that specifies which pages of memory can be accessed by each processor. We will use a bipartite graph representation for the relation S (see Fig. 2).

In order to describe groups of processors that can communicate with each other through one or more common pages we also introduce the *P-graph* (see Fig. 3) which is defined by the relation:

$$\mathbf{P} = \{(p_i, p_j)/ \text{ processors } p_j \text{ and } p_i \text{ can both access some page } m_h\}.$$

Let $(P_1, M_1), \cdots, (P_k, M_k)$ be the bipartite connected components of the *P-M* graph after a system failure has occurred. In order for the processors to reach a consensus on an appropriate configuration of the system, a communication page c_i must be selected for each component (P_i, M_i). In the reconfigured system the bipartite connected components (P_i', M_i') will satisfy the following conditions.

1) Every processor in P_i' can access c_i.

2) Every page in M_i' is accessible by all processors in P_i'.

Note that the second condition forces (P_i', M_i') to be a complete bipartite graph.

Every processor of the multiprocessor system is assumed to execute an infinite loop, alternating between application and reliability code. Moreover, when a processor detects a failure, it immediately starts executing the reliability code:

```
repeat
    on error goto reliability code;
    run application code;
    run reliability code;
forever
```

We assume that the execution time of the application code is much greater than the execution time of the reliability code. Thus, it is reasonable to assume that all faults occur during execution of the application code.

The reliability algorithms presented in the following sections are implemented as Pascal programs. Each processor will contain a copy of the relevant program in its local memory. In order to distinguish between those data structures stored in local and common memory we introduce the data types *commonmemory:*

type commonmemory = **array** [1..*m*] **of** commonpage;

and *localmemory*

type localmemory = **array** [1..*p*] **of** local page;

Both *commonpage* and *localpage* are "records" in the Pascal sense. They contain the information that is used by the algorithm under consideration and are thus defined differently in different sections of the paper. *CM* and *LM* are instances of *commonmemory* and *localmemory* respectively; *LM*[*i*] represents the local memory of the ith processor and $CM[i]$ is the ith common page.

Finally, we assume, as in PLURIBUS, the existence of a global clock, which is reliable and can be reliably read by all processors.

III. The Algorithm used in PLURIBUS

Management of system configuration and recovery functions on PLURIBUS is handled by the STAGE operating system [5]. The routines of the operating system are organized in levels or stages. The lowest level, or stage, is activated periodically, or when an error is detected. Each successive stage is enabled to run only if all earlier stages have successfully completed their system checks.

At one of the lowest stages of STAGE, a page in common memory has to be designated, through which the processors will communicate. The view taken by the PLURIBUS designers is that common memory is a more valuable resource than processors, since most applications require large global data structures. Thus they permit the extreme case, where all but one processor is turned off, as that one processor can access most of the common memory.

We consider what happens when the stage which establishes the communication page is activated. If a failure has occurred, different processors may now access different parts of common memory. A communications page c_i must be selected for each bipartite connected component (P_i, M_i) of the P-M graph. The PLURIBUS designers use the lowest indexed page in M_i as the communication page. The sets $P_i' M_i'$ in the reconfigured system are then derived by the conditions in Section II.

We present below the algorithm used by PLURIBUS and prove that it does indeed determine the communication page, as defined above. We then present an example showing that, in the case of multiple faults, the system reconfiguration induced by this algorithm does not follow either of the first two optimization criteria discussed in Section I.

Using the conventions described in Section II we define the local and global data structures used by the algorithm. Let p be the number of processors and m be the number of common memory pages.

Common memory pages will have the following structure:

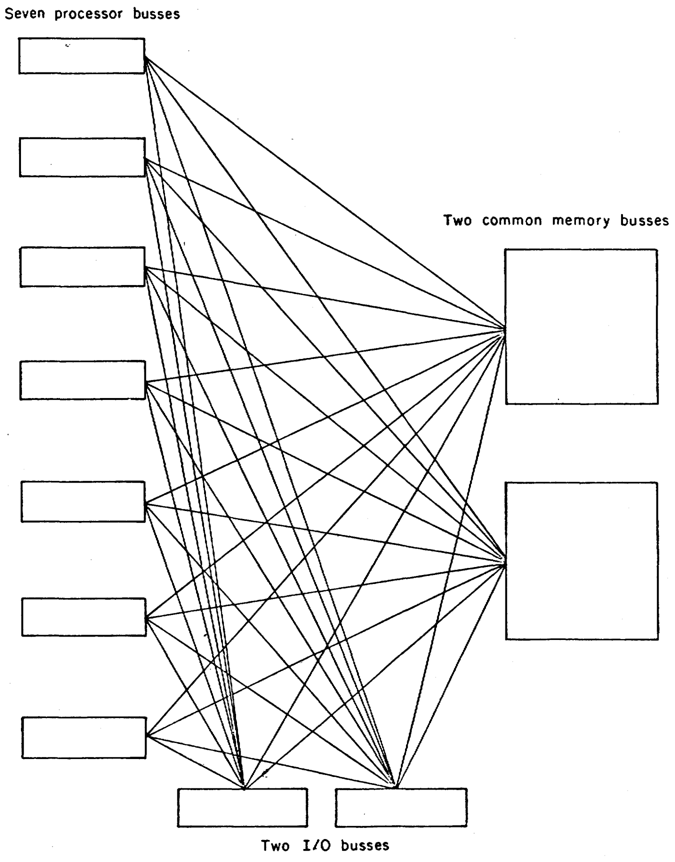


Fig. 1. The PLURIBUS architecture.

```
type commonpage = record commonptr: integer;
                         commontest: array [1..p] of boolean
                  end;
```

commonptr is a pointer to the communication page. *commontest* is a Boolean array whose jth entry is set to **true** if processor j wants to change the value of *commonptr* and is set to **false** if processor j agrees with the current value of *commonptr*.

During the first phase of the algorithm, each processor first determines as its candidate communications page, the lowest indexed common page that it can access. If processor i disagrees with the value of *commonptr* on a certain page, it sets the ith bit of *commontest* to **true** and goes to the second phase. If it agrees with the value of *commonptr*, it sets the whole *commontest* array to **false**. If it agrees with the value of *commonptr* on all pages that it can access, then it repeats the first phase of the algorithm a sufficiently large number of times to prevent the disagreeing processors from changing *commontest*.

The second phase of the algorithm is executed by each processor that wants to change the value of *commonptr* in some common page. The processor waits (sleeps) for some time (approximately one minute in PLURIBUS). Then it examines the associated value of *commontest*. If it is still true, the processor unilaterally corrects the value of *commonptr* in this page, provided that *commonptr* has not already been changed to a lower value. A processor executing the second phase will loop a sufficiently large number of times to make sure that it does not disagree with the value eventually assigned to *commonptr*. In Appendix I, we prove the correctness of the PLURIBUS algorithm under timing constraints which relate the number of iterations of the first and second phases of the algorithm to the execution times of these phases.

An obvious question is whether the algorithm maximizes the utilization of the two resources, processors and common memory, or any combination of them. The reader can easily verify that in the case of a *single* fault (a single bus or bus coupler fails) the number of common pages is maximized. The following example shows, however, that this is not true in the case of multiple faults. Consider the memory configuration

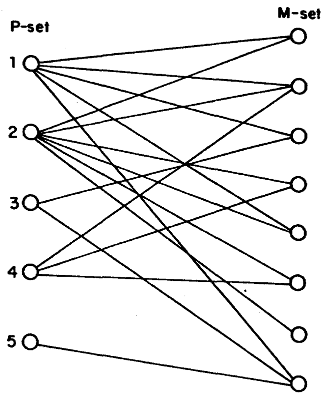


Fig. 2. A *P-M* graph.

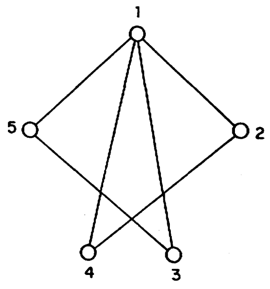


Fig. 3. Associated *P*-graph.

shown in Fig. 4: clearly, processors 1 and 2 have *mycandidate* = 3, while processors 3–7 have *mycandidate* = 5. The algorithm turns off processors 3–7. Neither pages nor processors are maximized. Maximization of the number of processors would result, if only the second bus were used by all processors. Common memory, on the other hand, would be maximized by turning off processors 1 and 2.

IV. General Algorithm for Propagating Information in Reliable Shared Memory Multiprocessor

A. Description of the Algorithm

In this section we present an algorithm for propagating local information associated with a particular processor to all other processors in the same connected component of the *P*-graph. In subsequent sections this algorithm is used to distribute information about the establishment of a communication page or system map. Our algorithm is conceptually similar to the algorithm for propagating local information regarding changes in the topology of a packet switching network that is discussed in [6]. However, in our model the synchronization of individual processors is considerably more complicated, since no message passing primitives are available prior to establishment of the communication page. In fact, an individual processor may not even know the identity of its neighbors in the *P*-graph. We assume that each processor has stored in its local memory an item of information that it would like to distribute to all the other processors; this item might be a list of the common pages that the processor can access or information about the communication page of a group of processors (see Section V).

In Appendix II, we describe the detailed structure of the data types *localpage* and *commonpage* that are used by our item propagation algorithm. Each *localpage* must provide storage for all the items of all processors. Initially, it contains only the item of its associated processor. Each *commonpage* must also provide temporary storage for all items; in addition, each will contain the variable *proc_num*, which gives the number of processors that can access the page and two synchronization variables *count1* and *count2*. Let *CM* and *LM* be instances of common and local memory, respectively.

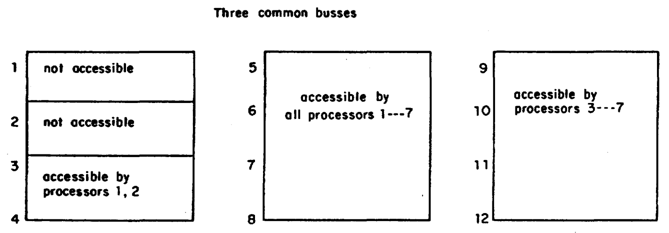

Three common busses
1
2
3
4
not accessible
not accessible
accessible by processors 1, 2
5
6
7
8
accessible by all processors 1---7
9
10
11
12
accessible by processors 3---7

Fig. 4. See example.

We next give a description of the distributed algorithm run by each processor. Every processor loops through four phases, labeled *P1* through *P4*, copying information from local memory to common memory and vice versa until no more items are copied in from the common memory. Finally, the processor announces termination by incrementing the value of a counter in all common pages that it can access.

```
initialize;
repeat
  P1: copy from local to common;
  P2: wait for all neighbors in the P-graph to finish P1;
  P3: copy from common to local;
  P4: wait for all neighbors in the P-graph to finish P3;
until finished;
announce termination;
```

Fig. 5 shows a possible execution of the algorithm for a multiprocessor system with four processors and six common pages. During the initialization phase, each processor increments the value of *proc_num* and *count2*, and zeros *count1* in all common pages that it can access. Thus, at the end of the initialization phase, in each common page the *proc_num* component will give the number of processors that can access the page, *count1* will be zero and *count2* will be equal to *proc_num*. After a timeout to ensure that each processor has finished, the processors begin copying information from local to common memory and then from common to local memory. The code for the four phases is given in Appendix II; an informal description is given as follows.

P1) Copy all ***new items*** to all accessible common pages; increase *count1* by one in all accessible common pages.

P2) Wait until in all accessible common pages *j*, *CM*[*j*]. *count1* = *CM*[*j*].*proc_num;* in all accessible common pages in which *count2* is ***still*** equal to *proc_num*, set *count2* to zero.

P3) Copy all ***new items*** from all accessible common pages; increase *count2* by one in all accessible common pages.

P4) Wait until in all accessible common pages *j*, *CM*[*j*]. *count2* = *CM*[*j*].*proc_num*; in all accessible com-

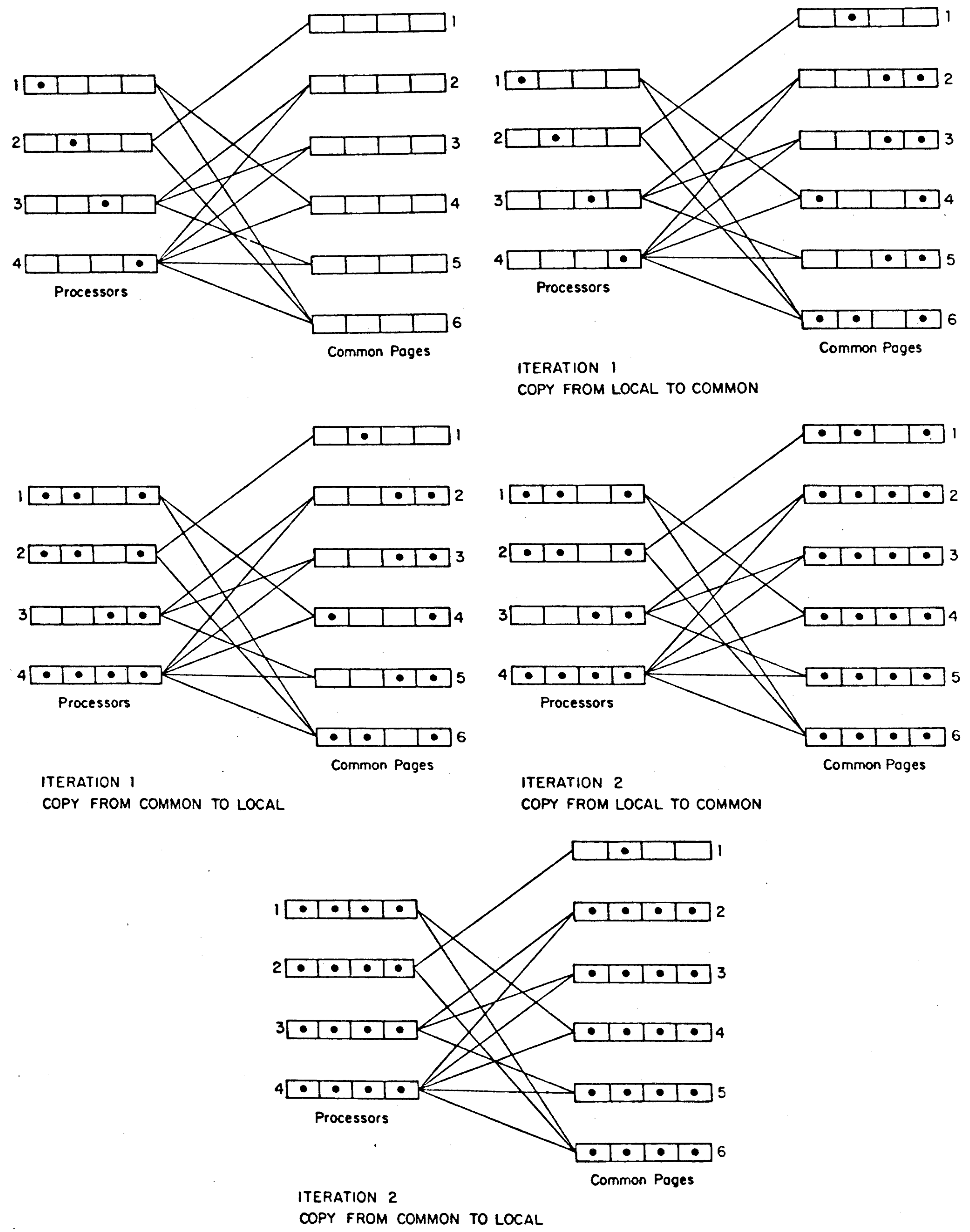

Processors
Common Pages
ITERATION 1
COPY FROM LOCAL TO COMMON
ITERATION 1
COPY FROM COMMON TO LOCAL
ITERATION 2
COPY FROM LOCAL TO COMMON
ITERATION 2
COPY FROM COMMON TO LOCAL

Fig. 5. Example execution of the general information propagation algorithm.

mon pages in which *count1* is ***still*** equal to *proc__num*, set *count2* to zero.

Phases *P1* through *P4* are repeatedly executed until no new items are brought in from the common memory in *P3*.

B. Analysis of the Algorithm

We use a global time frame to define the three functions **iteration, phase**, and **stage**. Function **iteration**(p_j, t) keeps track of how many times the **repeat** loop has been executed (see Section IV-A) by processor p_j at time t. **phase**(p_j, t) = *P1*, *P2*, *P3*, or *P4* depending on whether processor p_j is executing code in *P1*, *P2*, *P3*, or *P4* at time t. Finally, we define **stage** to be the function:

```
stage(p_j, t) ≜
  case
    $iteration(p_j, t) = i \wedge phase(p_j, t) = P1$   then $4i - 3$
    $iteration(p_j, t) = i \wedge phase(p_j, t) = p2$   then $4i - 2$
    $iteration(p_j, t) = i \wedge phase(p_j, t) = P3$   then $4i - 1$
    $iteration(p_j, t) = i \wedge phase(p_j, t) = P4$   then $4i$
  endcase
```

Lemma 1: Assume that processors p_j and p_k can both access some common page m_h. Then, for all t,

$$|stage(p_j, t) - stage(p_k, t)| \leq 3$$

Moreover, when processor p_j begins stage $4i + 3$, processor p_k must have finished stage $4i + 1$.

This lemma shows that processors which are neighbors in the P-graph can drift apart by at most three phases. Processors which are not neighbors in the P-graph may be considerably further out of phase.

The next lemma shows that the **repeat**-loops in phases $P2$ and $P4$ always terminate. Since these loops are the only places where a processor waits, this lemma effectively establishes the absence of deadlock.

Lemma 2: If processor p_j can access common page m_h and $phase(p_j, t) = P2(P4)$, with

$$CM[h].proc\_num > CM[h].count1 \ (CM[h].proc\_num > CM[h].count2)$$

then there will be a time $t' > t$ such that

$$CM[h].proc\_num = CM[h].count1 \ (CM[h].proc\_num = CM[h].count2)$$

and $phase(p_j, t') = P3(P1)$.

Proof: Given in Appendix II.

The next theorem assures that at the end of the ith iteration, processor p_j will know the *items* of all processors within distance i in the P-graph.

Theorem 1: If at time t $stage(p_j, t) = 4i$, then processor p_j will know (i.e., have stored in its local memory) all items of all processors p_k in the same connected component of the P-graph such that the distance (i.e., the shortest path) between p_j and p_k is less than or equal to i.

Proof: We will prove the theorem by induction on the number of iterations.

Basis: $iteration(p_j, t) = 1$ and *$stage(p_j, t) = 4$*. At $t' < t$, when $stage(p_j, t') = 3$, all neighbors of p_j will have stored in the common pages that they can access their private *items*. Hence, at time t, p_j will have stored in its local memory all *items* of processors at distance 1.

Induction Step: Assume that at time t, $stage(p_j, t) = 4i$. By the inductive hypothesis processor p_j knows the *items* of all processors at a distance i from p_j. By Lemma 2 there will be a time $t' < t$, when processor p_j begins execution of stage $4i + 3$. By Lemma 1 all neighbors p_k of p_j will have finished stage $4i + 1$. Hence, for each p_k there will be a time $t_k < t'$ when $stage(p_k, t_k) = 4i$. By the induction hypothesis p_k must know at time t_k the items of all processors within distance i. Thus, at time t' each p_k will have copied these values into common memory. At time t_j, when $stage(p_j, t_j) = 4i + 4$, p_j will have copied all new *items* from common memory and therefore know all *items* within distance $i + 1$. Q.E.D.

*Theorem 2: Let $C = \{p_1, \cdots, p_k\}$ be processors forming a connected component in the P-graph and $v_1, \cdots, v_k$ their private values (**items**). Then, upon termination of the algorithm the set $\{v_1, \cdots, v_k\}$ is stored in the local memories of all $p_1, \cdots, p_k$.*

Proof: Suppose that the **until**-clause evaluates to **true** at the end of the jth iteration. By inspection of $P1$ we note that at its end the whole local *update* array is set to **false**. Furthermore, at $P3$, for each ***new*** item copied from common to local (i.e., for each item k such that the local $already[k]$ is **false**), the corresponding *update* bit is set to **true**. Now, the only way that the **until**-clause can evaluate to **true** is for all of the bits of the *update* array to be **false**. It follows that during the jth iteration no *new* items have been copied from the common to the local memory of processors p_i. Hence, using Theorem 1, we can deduce that there are no processors at distance j from p_i, and thus p_i knows the items of all processors in C.

Q.E.D.

C. Complexity Issues

Because of Theorem 2, every processor p_i terminates its copy of the algorithm after $p^i_{\max}$ iterations, where $p^i_{\max}$ is the maximum length of all shortest paths between p_i and all other processors in the same connected component of the P-graph. It follows that there can be at most d iterations of the algorithm, where d is the diameter of (a connected component of) the P-graph. The next theorem specifies the minimum number of failures (memory, processor, or bus coupler) necessary to establish a P-graph with diameter d. It assumes that the P-vertices of the P-M graph represent processor buses and that the M-vertices of the P-M graph represent common memory buses. Each common memory bus consists of several common pages. An M-vertex is only deleted from the P-M graph when the associated common memory bus fails or when all common pages on the bus become inaccessible. Similarly, a P-vertex is deleted when either the bus fails or all processors on the bus die.

Theorem 3: Consider a complete P-M graph with p P-vertices and m M-vertices. Let f_{min} be the minimum number of faults required to achieve a P-graph of diameter d. Then $f_{min}(p, m, d) \triangleq$

if $d = 2$ **then** m

else if $2 < d \leq min(m, p - 1)$ **then** $m + p + d^2 - 3d - 1$.

Proof: Given in Appendix II.

It is clear from the above that even to augment the diameter from 1 to 2, a considerable number of faults is required (for $p = 14$ and $m = 10$, we need 10 coupler faults to get $d = 2$ and 23 faults to get $d = 3$!). We can thus realistically assume that no processor will have to iterate through the two copying phases of the algorithm more than once or twice. Assuming no contention when accessing common memory pages, we observe that the time complexity of the two copying phases is $O(pm)$.

V. Establishing a Communication Page

In this section we investigate optimal algorithms for determining a communication page. In Section V-A we present an algorithm which maximizes the number of processors in the reconfigured system. In Section V-B we describe an algorithm which maximizes the number of common pages that can be accessed by each processor in the reconfigured system. We show in Section V-C that the general problem of optimizing the value of an arbitrary performance function of both processors and pages is NP-complete, and in Section V-D we develop a probabilistic algorithm which yields a system configuration approximating the maximum for a special performance function.

A. Maximizing the Number of Processors

An obvious algorithm for maximizing the number of processors is to use the item propagation algorithm from Section IV to build a copy of the system map in the local memory of each processor. An individual processor will use the system map to determine the optimal choice for a communication page and shut itself off if it cannot access that page. This approach is inefficient for large multiprocessors where the system map would have to be represented by a large bit array. In the algorithm that we present below, it is only necessary to propagate the values of two integers.

Let $C = \{p_1, \cdots, p_k\}$ denote the set of processors in some connected component of the P-graph and let M be the set of pages accessible by the processors in C. Define PA_i to be the set of processors that can access common page $m_i \in M$. The cardinality of PA_i will be denoted by $|PA_i|$. Our goal is to determine a common page $m_i \in M$ for which $|PA_i|$ is maximum. If there is more than one page for which this is true, we select the lowest indexed one.

Our algorithm is a straightforward application of the information propagation algorithm. After the initialization phase and before starting iterating through the copying phases, every processor determines the page, among those that it can access, with the highest value of *proc_num* (*proc_num* again denotes, as in Section IV, the number of processors that can access a particular common page). This value together with the associated index of the common page is stored as the private *item* of every processor. After termination of the information propagation algorithm, every processor in C will have all the information needed to determine the page $m_i \in M$ for which $|PA_i|$ is maximum. For example, in Fig. 6 processors p_1, p_2, p_3, and p_4 will propagate the pair (2, 4) as their private *item* while processors p_5, p_6, and p_7 will propagate the pair (4, 3). If a processor's own *item* is different from $(m_{max}, |PA_{max}|)$ (where m_{max} is the lowest indexed common page for which $|PA_{max}|$ is maximum), the processor will shut down. All agreeing processors will be included in the reconfigured system. Common memory of the reconfigured system will consist of those pages accessible by all of the agreeing processors.

B. Maximizing the Number of Common Pages

Let $C = \{p_1, \cdots, p_k\}$ be the set of processors in some connected component of the P-graph. For every processor $p_i \in C$, let MA_i be the set of common pages that p_i can access. In order to maximize shared memory we must select a processor $p_h \in C$, such that $|MA_h|$ is maximal. If there is more than one such processor, we select the lowest indexed one.

The algorithm for determining p_h is fairly simple. Each processor p_i determines the number of pages that it can access and propagates this number to each of the other processors using the item propagation algorithm. Processor p_h will now be able to realize that $|MA_h|$ is maximum. To ensure that no processor which can access all pages in MA_h is shut off, processor p_h sequentially numbers all pages in MA_h. The highest numbered page is indicated by a special marker. After waiting for a reasonable amount of time, the other processors will shut down, if they cannot see all of the numbered pages in MA_h. By convention, the lowest indexed page in MA_h will be designated as the communications page. The code for this distributed algorithm is straightforward and will be left to the reader. In the example of Section V-A, our algorithm will correctly select page 5 as the communication page. The reconfigured system will consist of pages 5–12 and processors 3–7.

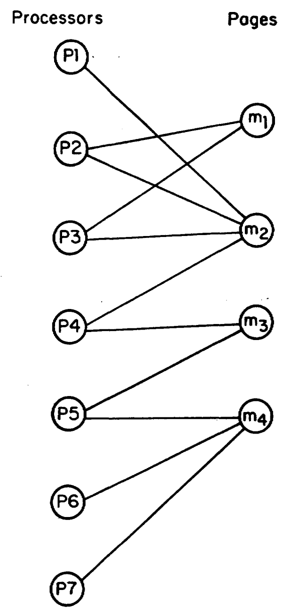


Fig. 6. Example interconnection for the algorithm 5.1.

C. Maximizing a General Performance Function

The algorithms in Sections V-A and V-B may result in unreasonable system configurations. Algorithm 5.1 might force all processors to run with only one page of memory, and algorithm 5.2 might permit only one processor to run on the whole common memory. A better tradeoff between computation speed and memory might be obtained by maximizing a function $f(p, m)$ which assigns a performance value to each configuration of p processors and m common pages in which every processor can access each memory page. Unfortunately, even if every processor has the system map stored in its local memory the problem of determining the optimal configuration, using a sequential algorithm, will probably require exponential time.

Theorem 4: The following problem is NP-complete.

"Given a P-M graph, a table defined function f and a constant c, is there a configuration with p processors and m common pages such that $f(p, m) > c$?"

Proof: The balanced complete bipartite subgraph (BCBS) problem [2].

"Given a bipartite graph $G = (V, E)$ and a positive integer $k \leq |V|$, are there two disjoint subsets $V_1, V_2 \subseteq V$

such that $|V_1| = |V_2| = k$ and such that $u \in V_1, v \in V_2$ implies that $\{u, v\} \in E$"

is NP-complete and can be reduced to our problem.

To reduce the BCBS to our problem, consider a graph G and a positive integer k. Consider also a function $f(|V_1|, |V_2|)$ such that

$$f(|V_1|, |V_2|) \triangleq$$
$$\textbf{if } |V_1| = |V_2| = k \textbf{ then } a > c$$
$$\textbf{else } b < c$$

where a and b are fixed integers and c is the constant in our problem. An algorithm to solve our problem for *P-M* graph G, performance function f and constant c, would solve the BCBS for graph G and constant k. Since this transformation can be made in polynomial time and our problem is in NP, it follows that our problem is NP-complete. Q.E.D.

D. A Probabilistic Algorithm for a Special Performance Function

A realistic assumption about the performance function is that it favors configurations (i.e., complete bipartite subgraphs of the *P-M* graph), where the ratio of the number of processors to the number of common pages is approximately equal to p/m where p is the number of processors and m is the number of common pages.

We show how the probabilistic algorithm proposed in [3] for finding cliques in general graphs can be modified to find complete bipartite subgraphs in a large *P-M* graph, which on the average enjoy the aforementioned property. The modified algorithm assumes that each processor has access to the entire system map. The algorithm in [3] works as follows.

Let G be the input graph and let $\{l_1, l_2, \cdots\}$ be a set of distinct labels. We randomly number the vertices of G and associate label l_1 with vertex 1. We examine the remaining vertices in increasing order, associating label l_j with vertex i $(2 \leq i \leq n)$ if j is the least integer such that all vertices labeled l_j are joined (in G) to i. The output of the algorithm is one of the resulting cliques (for example, the biggest one) and can be expected to be half the size of the maximal clique [3].

The *P-M* graph cannot be directly used as input for the algorithm since a bipartite graph does not have any cliques (except for trivial 2-vertex ones). By adding dummy edges it is possible to temporarily mask this property of the *P-M* graph from the algorithm, while, at the same time, maintaining the random graph model used for the analysis of the algorithm in [3]. A random graph, as defined in [3], is a graph G with vertex set $\{1, 2, \cdots, n\}$ and a set of edges E such that the probability that an edge connects any two vertices of G is q and is independent of the probability that any other pair of vertices is connected with an edge. Let q be the probability with which a processor is connected to a page in the *P-M* graph. We connect the *P*-vertices with each other and the *M*-vertices with each other with probability q, thus obtaining a random graph R. The algorithm can now be applied to R, to obtain a clique C. By deleting the dummy edges from C a complete bipartite subgraph of the *P-M* graph will be obtained.

Theorem 5: Let C be the clique formed by the algorithm on input graph R. Let s be the number of P-vertices and t be the number of M-vertices in C. Also, let $\lambda = p/m$ where p is the number of P-vertices and m is the number of M-vertices, in the original P-M graph. Then

$$E(s)/E(t) = \lambda \text{ and } \sigma(s)/\sigma(t) = 1$$

Proof: s and t are random variables following the hypergeometric distribution (see Drake [1]). Thus, we have

$$E(s) = p(s + t)/(p + m)$$
$$E(t) = m(s + t)/(p + m)$$

So,

$$E(s)/E(t) = p/m = \lambda.$$

Also

$$\sigma(s) = \sigma(t)$$
$$= p\,m\,(s + t)(p + m - s - t)/(p + m)^2(p + m - 1)$$

So,

$$\sigma(s)/\sigma(t) = 1.$$ Q.E.D.

Note that the independence assumption in [1] will be violated by failure of an entire memory bus. In order for our analysis to be realistic, the probability of a memory bus failure must be negligible. Alternatively, the algorithm could be applied to a *P-M* graph in which the *M*-vertices correspond to memory buses rather that individual memory pages.

VI. Conclusion and Open Problems

Algorithms have been presented in this chapter for reconfiguring shared memory multiprocessor systems under various optimization criteria. We are currently developing an analytical model for the behavior of the item propagation algorithm which will permit accurate estimates of execution time as a function of system parameters such as number of processors and number of pages of common memory.

Another area of current research is the development of appropriate optimization criteria for various classes of multiprocessor applications. It seems likely that either processors alone or memory alone will be insufficient for most applications. The results of Sections V-C and V-D show that approximate or probabilistic algorithms will be useful in finding optimal system configurations whenever the performance criterion is nontrivial. Finally, we are attempting to extend our algorithms, using techniques similar to those discussed in [4], so that malfunction (i.e., erratic behavior) of processors and/or common memory is allowed.

Appendix I

Analysis of the PLURIBUS Algorithm

Processor i maintains the following variables in its local memory.

1) A Boolean array called *mycmmap*$[i]$ whose jth entry is **true** if processor i can access the jth common page and is **false** otherwise.

2) An integer variable *mycandidate* which is assigned the minimum page index among all indexes of pages which can be accessed by processor i:

$$mycandidate = min\{j/mycmmap[j] = \textbf{true}\}$$

3) A Boolean variable *next* which is initially **false** and becomes **true** iff processor i wants to execute the second phase of the algorithm.

4) An integer variable *oldcommonptr* which is assigned the old value of *commonptr*, whenever processor i wants to change the value of *commonptr*.

Thus local pages have the structure

```
type local page = record mycmmap:
                         array [1..m] of boolean;
                           mycandidate: integer;
                           next: boolean;
                           oldcommonptr: integer;
                         end;
```

As in Section II let *CM* and *LM* be instances of type *commonmemory* and *localmemory*, respectively:

```
first__phase:   with LM[me] do
                  begin next := false;
                    for k := 1.. num__iterations1 do
                    begin
                      forall i := mycandidate..m
                             such that mycmmap[i] do
                      begin with CM[i] do
                        if commonptr = mycandidate
                        then for j := 1..p do
                                commontest[j] := false
                       else begin
                         oldcommonptr := commonptr;
                         commontest[me] := true;
                         next := true;
                      end;
                      if next then goto second__phase;
                    end;
                  end.
```

Some notation in the code above needs explanation; in order to shorten notation and increase readability we introduce the control structure:

```
forall ⟨index-var⟩ := ⟨exp1⟩..⟨exp2⟩
             such that ⟨boolean exp⟩ do ⟨for-body⟩;
```

with the following semantics: for every value of the index variable starting from *⟨exp1⟩* with increments of 1, the value of the Boolean expression is evaluated. If the result is **true**, then the *⟨for-body⟩* is executed. If the result if **false**, then the *⟨for-body⟩* is not executed for this value of the index variable.

We assume that assignment statements are indivisible atomic actions. We use the notation "⟨compound statement⟩" to indicate that "compound statement" is indivisible and must be executed without interruption:

```
second__phase:      sleep(time);
                    with LM[me] do
  for j := 1..num__iterations2 do
    forall i := mycandidate..m
           such that mycmmap[i] do
      with CM[i] do
         ⟨if commontest[me]
                  and
            ((oldcommonptr = commonptr)
                  or
            ((oldcommonptr ≠ commonptr)
                  and
            (mycandidate ≤ commonptr)))
      then commonptr := mycandidate
      else turnoff(me)⟩;
```

The indivisibility of the conditional in the code above, is required to avoid a race condition, where two processors with different values of *mycandidate* (both different from *oldcommonptr*) both alter the value of *commonptr*, but neither is turned off. This can be easily implemented as a test-and-set operation on the vaiables *commontest*[*me*] and *commonptr*.

In order to prove the correctness of the algorithm and analyze its performance we make the following assumptions.

1) When the algorithm begins executing, the values of *commonptr* in all pages are consistent, i.e., they all point to the same page. This page may not be accessible by some or all processors, however.

2) The following time constants have been accurately determined from empirical data.

a) $Dt_{1,max}$ is the time required by each processor to run the first phase of the algorithm once (in general, the first phase can be executed *num__iterations1* times). Observe that $Dt_{1,max}$ is difficult to determine, as there may be several processors executing the first phase. These processors will compete for use of the common pages and will consequently be delayed while waiting for each other. Hence $Dt_{1,max}$ can best be thought of, as the maximum time for a single processor to execute the first phase once, under worst case scheduling conditions. $Dt_{2,max}$ is similarly defined for the second phase.

b) $Dt_{1,min}$ is defined to be the minimum time required by a single processor to execute the first phase once, assuming the best possible scheduling conditions (i.e., the processor never waited while trying to access a common page). $Dt_{2,min}$ is similarly defined.

c) D is the maximum initial delay for each processor before starting the execution of the algorithm, given that at least one processor started.

d) Finally, *time* is the sleeping time of each processor waiting to execute the second phase of the algorithm.

Call a processor an *agreeing* processor, if it agrees with the values of *commonptrs* on all pages it can access; otherwise call it a *disagreeing* processor. Now, define the following relation on the set of processors $P = \{p_1, \cdots, p_n\}$

$$\boldsymbol{R} = \{(p_i, p_j)/ \textit{ the values of mycandidate are equal for } p_i \textit{ and } p_j\}.$$

$\boldsymbol{R}$ is an equivalence relation. Let $C_1, \cdots, C_k$ be its equivalence classes. By convention we assume that if $i < j$ the value of *mycandidate* for all processors in C_i will be less than the corresponding value for all processors in C_j. C_1 will be the equivalence class of all agreeing processors (if any). Let $N_1, \cdots, N_k$ be sets defined as follows:

N_i = *{j/common page j can be seen by some processor in* C_i}.

The equivalence classes $C_1, \cdots, C_k$ correspond to cliques in the P-graph, since for any two processors p_i and p_j in C_h there is at least one page that they can both access, namely the one pointed to by *mycandidate*. When processor p_i is turned off a new P-graph is formed such that node p_i is deleted as well as all edges incident to it. Thus, the algorithm reduces the equivalence classes $C_1, \cdots, C_k$ to new sets $C_1', \cdots, C_k'$, some of which may be empty.

Theorem 6: Assume that prior to the execution of the algorithm

1) the values of all **commonptrs** *are consistent in all common pages, i.e., they all point to the same common page, and*

2) all memory faults have already occurred, and no faults will occur during execution of the algorithm

$$\textbf{num\_iterations1} \geq (D + Dt_{1,max})/Dt_{1,min}$$

$$D + Dt_{1,max} \leq Dt_{1,min} + \textbf{time}$$

$$\textbf{num\_iterations2} \geq$$

$$(D + Dt_{1,max} - Dt_{1,min} + Dt_{2,max})/Dt_{2,min}.$$

Then, after execution of the algorithm:

1) if a processor $p_i \in C_i$ *was connected in the P-graph to some* $p_j \in C_j$ *such that* $j < i$, *then* p_i *will be turned off, and*

2) in every N_i' *such that* C_i' *is nonempty, all pages have the same value for* **commonptr**.

Proof:

1) Consider a processor $p_i \in C_i$ connected in the P-graph to a processor $p_j \in C_j$, such that $j < i$. We distinguish two cases.

Case 1: Assume p_j is an agreeing processor, i.e., $C_j = C_1$. Then p_j will start looping on the first phase of the algorithm for a *num_iterations1* times, while p_i will exit the first phase (as *next* becomes **true**) and sleep for *time* units of time. We have to make sure that the following two events never occur:

a) p_i, being very fast, immediately starts executing the first phase, sleeps for a while, and then executes the second phase, without giving p_j a chance to execute the first phase. To ensure that this does not happen, it should be the case that

$$D + Dt_{1,max} \leq Dt_{1,min} + time$$

which is an assumption of the theorem.

b) p_i, being very slow, begins execution of the algorithm with a delay of D units of time, executes the first phase in $Dt_{1,max}$ units of time and then goes to sleep. However, p_j, which entered immediately, looped through the first phase for less than $D + Dt_{1,max}$ units of time, thus not being able to reset the jth bit of *commontest*, set by p_i. To avoid this event, it should be the case that

$$num\_iterations1 \times Dt_{1,min} \geq D + Dt_{1,max}$$

which is an assumption of the theorem.

Case 2: p_i is not connected to an agreeing processor in the P-graph. It will thus execute the second phase and will try to change the value of *commonptrs*. However, p_i is connected to a p_j in the P-graph such that $j < i$. Hence we must prove that p_i will eventually turn itself off, or equivalently that the following two events will never occur:

a) p_j, being too fast, will first change the value of *commonptr*. In this case, however, p_i will turn itself off, as the condition in the second phase of the algorithm is evaluated to **false**.

b) p_j, being too slow, will change the value of *commonptr* after p_i stops looking on the second phase. For this event not to occur, it should be the case that

$$D + Dt_{1,max} + TIME + Dt_{2,max}$$

$$\leq Dt_{1,min} + TIME + num\_iterations2 \times Dt_{2,min}$$

which reduces to the last assumed inequality of the theorem.

2) Suppose that there were a page r in N_i whose value of the *commonptr* is not equal to that of *mycandidate* for all processors in C_i', where C_i' is nonempty. Call the former value v_1 and the latter v_2. We distinguish two cases.

Case 1: $v_1 < v_2$. Then, there should be a processor p_k with *mycandidate* = v_1 which is connected in the P-graph with a processor p_i in C_i' that can access r. This processor belongs to a class C_j' such that $j < i$ and by the first part of the theorem, it should have turned processor p_i off, a contradiction.

Case 2: $v_1 > v_2$. Again, a processor p_k should exist with *mycandidate* = v_1 which is connected in the P-graph with a processor p_i in C_i' that can access r. This processor belongs to a class C_j' such that $j > i$. By the first part of the theorem p_k should have been turned off by p_i. By inspection of the code for the second phase, it is clear that p_k will turn itself off, only if it disagrees with the value of *commonptr* in page r, a contradiction. Q.E.D.

Appendix II

Proofs Related to the Information Propagation Algorithm of Section IV

A common memory page is here defined as follows:

```
type commonpage = record info:        array [1..p]
                                        of item;
                         update:      array [1..p]
                                        of boolean;
                         proc_num:    integer;
                         count1:      integer;
                         count2:      integer;
                         termcount:   integer;
                  end;
```

Entry *info*[*i*] stores the item local to processor *i*; *item* might be a list of the common pages that a particular processor can access or information about the communication page of a group of processors (see Section V). The *update* component is introduced for efficiency reasons: a processor will never need to copy to local memory any entry in the *info* array for which the corresponding *update* bit is false. The number of processors that can access the page is stored in *proc__num* and is determined during the initialization phase of the algorithm. Finally, *count1* and *count2* are synchronization variables and *termcount* is a counter used to announce termination.

Local pages will have the following structure:

```
type local page = record   info:       array [1..p]
                                         of item;
                           already:    array [1..p]
                                         of boolean;
                           update:     array [1..p]
                                         of boolean;
                           mycmmap:    array [1..m]
                                         of boolean;
                              done:          boolean;
                              finished:      boolean;
                  end;
```

Here, the *info* and *update* components are used as in a *commonpage*. The bits of *already* are set whenever an *item* is received in the local memory. The Boolean array *mycmmap* has its *i*th bit set if the processor owning the *localpage* can access the *i*th common page. The Boolean *done* is used to signal the end of a copying phase and the Boolean *finished* signals the end of all copying phases.

Let *CM* and *LM* be instances of common and local memory, respectively.

During the initialization phase the system function *setup* sets the bits of the *mycmmap* array corresponding to pages accessible to processor "*me*." Function *initialize* puts the private *item* in the local *info* array. The number of active processors able to access a particular page is assigned to *proc__num*. Every processor increments the value of *proc__num* and of *count2* (initial values are zero; at the end of the algorithm they are reset to zero), and it zeros *count1*, for all pages it can see and then waits for a reasonable time until all processors finish. Finally, the appropriate bit of the *update* array is set to **true** to indicate that local *info*[*me*] contains new information which has to be copied to common memory:

```
with LM[me] do
  begin
    setup (mycmmap); initialize (info[me]);
    forall j := 1..m such that mycmmap[j] do
      with CM[j] do
        begin
          proc__num := proc__num + 1;
          count1 := 0; count2 := count2 + 1;
        end;
    sleep(time);
    update[me] := true; already[me] := true;
  end;
```

Next, every processor begins copying information from local to common memory and then from common to local memory.

In Phase *P1*, information is copied from the local to the common memory, *finished* is tentatively set to **true**; but it becomes **false** in *P3* if new *items* are copied into local memory. Every processor selects the next page to access in a random manner so as to minimize contention with other processors. This random selection is implemented through the two (local) functions *empty* and *getnext*. *empty* is a Boolean function which returns **true** if the calling processor still has pages to access. *getnext* is a function returning the index of the next page to be accessed. This page is randomly selected among those pages which are accessible but have not yet been accessed by the calling processor. After selecting a page the processor copies from local to common memory all new *items* in its local memory, and increases the value of the synchronizaiton variable *count1* by one:

"copy from local to common" stands for

```
P1: LM[me].finished := true;
    while not (empty(me)) do
      begin
        current := getnext(me);
        with CM[current] do
          begin
            forall k := 1..p such that
                                 LM[me].update[k] do
              begin
                info[k] := LM[me].info[k];
                update[k] := true;
              end;
            count1 := count1 + 1;
          end;
      endwhile;
    clear (LM[me].update);
```

In phase *P2*, the processor waits until all processors which access at least one of the pages it can access, are finished with phase *P1*. It then initializes the second synchronization variable *count2* to zero, provided that no other processor has already done so. Note that this must be an indivisible test-and-set operation to avoid the situation where two processors realize that *CM*[*j*].*proc__num* = *CM*[*j*].*count2*, one of them sets *count2* to zero, proceeds to *P3*, increases *count2* by one, and then the other processor wakes up and incorrectly resets *count2* to zero:

```
P2: with LM[me] do
      begin
        repeat
          done := true;
          forall j := 1..m such that mycmmap[j] do
            done := done and
                  (CM[j].count1 = CM[j].proc__num);
        until done;
        forall j := 1..m such that mycmmap[j] do
          ⟨if CM[j].proc__num = CM[j].count2
                then CM[j].count1 := 0⟩;
      end;
```

In phase *P3*, copying from common to local memory is performed. This phase is symmetric to *P1*, except for the fact that the processor does not recopy into its local memory *items*, which were already there:

"copy from common to local" stands for

```
P3: while not (empty(me)) do
```

```
    begin
    current := getnext(me);
      with CM[current] do
        begin
          for k := 1..p do
          if update[k] and not
                         (LM[me].already[k]) then
           begin
             LM[me].info[k] := info[k];
             LM[me].update[k] := true;
             LM[me].already[k] := true;
             LM[me].finished := false;
           end;
          count2 := count2 + 1;
        end;
      endwhile;
```

Finally phase *P4* is similar to phase *P2*. Processor *i* waits until all other processors accessing common pages that *i* can access, have finished copying:

```
P4: with LM[me] do
      begin
        repeat
          done := true;
          forall j := 1..m such that mycmmap[j] do
            done := done and
                  (CM[j].count2 = CM[j].proc_num);
        until done;
        forall j := 1..m such that mycmmap[j] do
          begin
            ⟨if CM[j].proc_num = CM[j].count1
                      then CM[j].count1 := 0⟩;
            clear (CM[j].update);
          end;
      end;
```

Phases *P1* through *P4* are repeatedly executed until the Boolean variable *finished* remains **true** throughout execution of *P1*, *P2*, *P3*, *P4*, i.e., until no new items are brought in from the common memory in *P3*. A termination phase is then executed, where the processor increases the value of the counter *termcount* in all accessible common pages. It then waits until the value of *termcount* becomes equal to *proc_num* in all accessible common pages. Finally it clears the parts of local and common memory that are used by the information propagation algorithm.

```
  while not empty(me)) do
    begin
      current := getnext(me);
      with CM[current] do termcount := termcount + 1;
    end;
  with LM[me] do
    repeat
      done := true;
      forall j := 1..m such that mycmmap[j] do
        done := done and
                (CM[j].count1 = CM[j].proc_num);
    until done;
  clear(LM[me]); clear(CM);
```

Proof of Lemma 2: Assume the contrary, i.e., assume that at all times $t' \geq t$, $phase(p_j, t') = P2(P4)$ and

$$CM[h].proc\_num > CM[h].count1 \; (CM[h].proc\_num > CM[h].count2)$$

Let $stage(p_j, t') = 4i - 2(4i)$. As processor p_j cannot start stage $4i - 1(4i + 1)$ (or equivalently phase $P3(P1)$ of the $i(i + 1)$th iteration), there must exist, by Lemma 1 at least one processor p_k which has not yet finished stage $4i - 3$ $(4i - 1)$ (or equivalently phase $P1$ $(P3)$ of the ith iteration). If processor p_k is already in stage $4i - 3$ $(4i - 1)$, nothing will prevent it finishing it, thereby increasing the value of *count1* (*count2*) by one and making it equal to *proc_num in* page m_h. Thus, p_k must be waiting in stage $4i - 4$ $(4i - 2)$, because, in some page m_n,

$$CM[n].proc\_num > CM[n].count2 \; (CM[n].proc\_num > CM[n].count1)$$

Moreover, m_n must be different from m_h; if they were equal, p_j could not possibly have progressed to a higher stage than p_k, contrary to our hypothesis. By a similar argument, we can find another processor p_m waiting in stage $4i - 6$ $(4i - 4)$, for which p_k is waiting. We can thus construct a chain of (pairwise distinct) processors $(p_j =)p_{i_1}p_{i_2} \cdots p_{i_k}$ and a chain of (pairwise distinct) common memory pages $(m_h =)m_{j1}m_{j2} \cdots m_{jk}$ such that for all $t' > t$:

$$stage(p_{i_n}, t') = 4i - 2n \; (stage(p_{i_n}, t') = 4i - 2(n - 1)), n \geq 1$$

$$CM[j_n].proc\_num > CM[j_n].count1$$
$$(CM[j_n].proc\_num > CM[j_n].count2)$$

for r odd, and

$$CM[j_n].proc\_num > CM[j_n].count2$$
$$(CM[j_n].proc\_num > CM[j_n].count1)$$

for r even.

Observe, however, that these two chains must be finite. In fact, both chains cannot be longer than $2i(2i + 1)$ since for all i_j processor $p_{i_{j+1}}$ is waiting at a stage, which is two stages lower than the stage where p_{i_j} is waiting. Thus, we distinguish three cases.

1) $min(p, m, 2i - 1) = p$: Processor p_{i_k} does not have to wait for any processor and can proceed—a contradiction.

2) $min(p, m, 2i - 1) = m$: The length of the chain of common memory pages must be less than m, since the pages in the chain are distinct. Thus p_{i_k} will be able to proceed—a contradiction as in the first case.

3) $min(p, m, 2i - 1) = 2i - 1$: Processor p_{i_k} is just starting execution of the algorithm and does not have to wait—also a contradiction.

In each case, we conclude that our initial assumption was false and that there is a time $t' > t$ such that

$$CM[h].proc\_num = CM[h].count1$$
$$(CM[h].proc\_num = CM[j].count2)$$

Q.E.D.

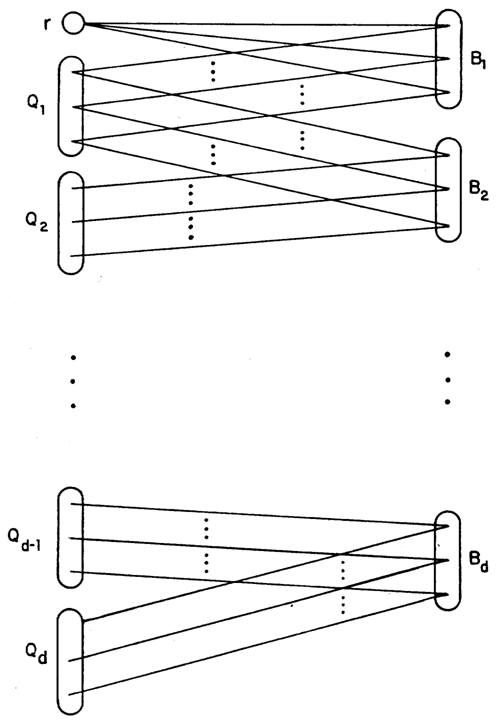


Fig. 7. For the proof of the complexity theorem.

Proof of Theorem 3: Let Q be the P-graph of diameter d, obtained from the original complete P-graph by the minimal number of faults. Let r be a processor bus which is at distance d from some other processor bus in the graph. Let $Q_i (0 \le i \le d)$ be the set of processor buses at distance i from r and let q_i be the cardinality of Q_i (thus, $Q_0 = \{r\}$ and $q_0 = 1$). By our assumption on the minimality of faults we see that every processor bus in Q_i should be connected to all processor buses in Q_{i+1} for $0 \le i \le d - 1$. Let G be the P-M graph corresponding to Q and let $B_i (1 \le i \le d)$ be the set of common memory buses connecting Q_{i-1} and Q_i *in* G (see Fig. 7). Denote the cardinality of B_i by b_i. By the same minimality argument, each common memory bus in B_i must be connected to such processor bus in Q_{i-1} and Q_i.

Let f_m be the number of common memory bus faults, f_p the number of processor bus faults and f_c the number of bus coupler faults, required to obtain G. The following inequalities obviously hold:

$$q_1 + \cdots + q_d = p - f_p - 1, \qquad 0 \le f_p \le p - d - 1$$
$$b_1 + \cdots + b_d = m - f_m, \qquad 0 \le f_m \le m - d$$

We consider first the special case where $d = 2$. In this case

$$q_1 + q_2 = p - f_p - 1, \quad b_1 + b_2 = m - f_m$$

and the number of bus coupler faults is

$$f_c = b_1 + q_2 b_2.$$

Thus the total number of faults is

$$f(q, b, 2) = f_p + f_m + f_c = f_p + f_m + b_1 + q_2 b_2.$$

This expression is minimized for $f_p = 0, f_m = 0, q_2 = 1$; thus $f_{min}(q, b, 2) = b$.

Next, we consider the case where $d > 2$. Let

$$S_i = b_1 + b_2 + \cdots + b_{i-1} + b_{i+2} + \cdots + b_d (1 \le i \le d)$$

The number of bus coupler faults required to reduce the complete P-M graph to G is given by

$$f_c = (b_2 + \cdots b_d) + q_1 S_1 + q_2 S_2 + \cdots + q_{d-1} S_{d-1} + q_d b_1 + q_d S_d.$$

We claim that the following assignment of values minimizes $f(q, b, d)$:

$$f_m = m - d, \quad f_p = p - d - 1$$
$$b_1 = \cdots = b_d = 1, \quad q_1 = \cdots = q_d = 1.$$

With this assignment, the number of bus coupler faults is

$$f_c = d(d - 1)$$

and the total number of faults is

$$f_{min}(q, b, d) = m + p + d^2 - 3d - 1.$$

Thus, we have to prove that

$$f(q, b, d) \ge f_{min}(q, b, d).$$

First observe that $S_i \ge d - 2$. Hence,

$$\begin{aligned} f(q, b, d) &= f_p + f_m + f_c \\ &\ge f_p + f_m + b_2 + \cdots + b_d + q_d b_1 \\ &\quad + (d-2)(p - f_p - 1 - q_d) + q_d(d-2) \\ &\ge m + p(d-2) - f_p(d-3) - (d-2). \end{aligned}$$

Since $f_d \leq p - d - 1$, it follows that

$$f(q, b, 2) \geq m + p + d^2 - 3d - 1. \qquad \text{Q.E.D.}$$

Acknowledgment

The authors wish to acknowledge the help of O. Shmueli, J. Reif, J. Robinson, and E. Roberts.

References

[1] A. Drake, *Fundamentals of Applied Probability Theory*. New York: McGraw-Hill, 1967.

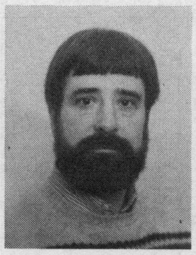
[2] M. R. Garey and D. S. Johnson, *Computers and Interactibility: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.

[3] G. R. Grimmet and C. J. H. McDiarmid, "On coloring random graphs," *Proc. Camb. Phil. Soc.*, vol. 77, p. 313, 1975.

[4] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. Ass. Comput. Mach.*, vol. 27, p. 2, Apr. 1980.

[5] J. G. Robinson and E. S. Roberts, "Software fault-tolerence in the Pluribus," in *Proc. Nat. Comput. Conf.*, 1978.

[6] W. D. Tajibnapis, "A correctness proof of a topology information maintenance network," *Commun. Ass. Comput. Mach.*, vol. 20, July 1977.



Edmund M. Clarke (M'80) received the B.A. degree in mathematics from the University of Virginia, Charlottesville, in 1967, the M.A. degree in mathematics from Duke University, Durham, NC, in 1968, and the Ph.D. degree in computer science from Cornell University, Ithaca, NY, 1976.

After leaving Cornell, he taught in the Department of Computer Science, Duke University, for two years. In 1979 he moved to Harvard University, Cambridge, MA, where he is currently an Assistant Professor of Computer Science in the Division of Applied Sciences. His interests include distributed systems, programming language semantics, and theory of computation.

Dr. Clarke is a member of the Association for Computing Machinery, Sigma Xi, and Phi Beta Kappa.

Christos N. Nikolaou received the diploma in electrical and mechanical engineering from the National Technical University of Athens, Greece, in 1977 and the M.S. and Ph.D. degrees in applied mathematics and computer science from Harvard University, Cambridge, MA, in 1979 and 1982 respectively.

In October 1981, he accepted a postdoctoral position at the IBM T. J. Watson Research Center where he is currently conducting research on network reliability problems.

Some New Results About the (d, k) Graph Problem

GERARD MEMMI AND YVES RAILLARD

Abstract—**The (d, k) graph problem which is a still open extremal problem in graph theory, has received very much attention from many authors due to its theoretic interest, and also due to its possible applications in communication network design. The problem consists in maximizing the number of nodes n of an undirected regular graph (d, k) of degree d and diameter k. In this paper, after a survey of the known results, we present two new families of graphs, and two methods of generating graphs given some existing ones, leading to further substantial improvements of some of the results gathered by Storwick [21] and recently improved by Arden and Lee [3] and also by Imase and Itoh [11].**

Index Terms—**Communication network, diameter minimization, (d, k) graph, graph generating operations, graph theory, Moore graph.**

Manuscript received February 26, 1980; revised July 20, 1981 and February 1, 1982.

G. Memmi is with the Central Research Laboratory, Thomson-CSF, Orsay Cedex, France.

Y. Raillard is with Thomson-CSF Telephone, Boulogne-Billaucourt, France.

I. Introduction

Rapid advances in very large scale integrated circuit technology have stimulated a great interest in microprocessor networks. In such networks, minimizing the distance between every couple of microprocessors obviously leads to more efficient communication networks (reducing delays and load on the lines). Furthermore, the cost of the interconnection among the microprocessors increases with the number of physical lines between two microprocessors in the network.

In graph theoretic terms microprocessors are called nodes; lines edges and the network are then modeled by an undirected graph. In this context, the main practical problem to be solved can be stated as follows: given a number of nodes, interconnect them minimizing both the number of edges and the distance between nodes. Conversely, given a maximum degree d (i.e., the number of edges incident at a node), and a maximal distance k find a graph of maximal order. This last formulation is known as the (d, k) graph problem [6].