RAIRO. INFORMATIQUE THÉORIQUE ET APPLICATIONS

G. GEORGAKOPOULOS
A. STAFYLOPATIS

An approach to parallel algorithm design

RAIRO. Informatique théorique et applications, tome 27, n° 2 (1993), p. 85-95

<http://www.numdam.org/item?id=ITA_1993__27_2_85_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 27, n° 2, 1993, p. 85 à 95)

AN APPROACH TO PARALLEL ALGORITHM DESIGN ([1]) (*)

by G. GEORGAKOPOULOS and A. STAFYLOPATIS ([2])

Abstract. – *Effort dedicated to the analysis of parallel computation has mainly focused in the direction of finding parallel algorithms for specific problems. On the other hand, there is a need for general purpose tools for "automatically" assigning complex tasks to parallel computing machines (compilers for parallel machines provide an example). In this paper, we describe an approach for finding an optimal, in some sense, schedule for assigning tasks to a set of parallel processors. Finding this schedule remains an NP-complete problem in the general case, but our definition allows us to solve it in polynomial time for a large class of task graphs and a large class of machine architectures. Point symmetric graphs and their generalization play an important role, and some of their properties are discussed.*

Résumé. – *L'effort dédié à l'analyse du calcul parallèle a surtout visé la recherche d'algorithmes parallèles pour des problèmes particuliers. Cependant, il est nécessaire de développer des outils généraux pour l'allocation « automatique » de tâches complexes à des machines parallèles (voir par exemple le cas de compilateurs pour machines parallèles). Nous présentons ici une approche qui permet de trouver une politique optimale, dans un sens, pour allouer des tâches à un ensemble de processeurs parallèles. Il s'agit d'un problème NP-complet au cas général, mais notre définition permet de le résoudre en temps polynomial pour une large catégorie de graphes de tâches et pour une large catégorie d'architectures. Les graphes symétriques et leur généralisation jouent un rôle important et nous en discutons certaines propriétés.*

1. INTRODUCTION AND BASIC DEFINITIONS

Graphs have been largely used so far in modeling the two fundamental notions in computer science: the *computation* and the *machine*.

We consider the computation as a complex job composed of simpler *tasks*, which are represented as the vertices of a graph $G=(V, E)$. The edges of the

(*) Received February 1991, accepted July 1992.
([1]) This work was partly supported by the General Secretariat of Research and Technology of Greece under Grant No. 9707. First author's current address is: Aegean University, Department of Mathematics, Karlovassi, 832 00 Samos, Greece.
([2]) Computer Science Division, Department of Electrical and Computer Engineering, National Technical University of Athens, 157 73 Zographou, Athens, Greece.

Informatique théorique et Applications/Theoretical Informatics and Applications
0988-3754/93/02 85 11/$3.10/

graph are determined by the relation $\alpha \to \beta$, defined between tasks, which is used to denote that task α is prerequisite for task β, *i.e.*, task β may begin execution only after the completion of task α. A basic attribute of a task is its *duration*, a natural number denoting the time units necessary for the execution of the task. The graph is considered to be equivalent to a partial ordering, *i.e.*, to be directed and acyclic.

These tasks are processed by a machine consisting of a number of interconnected processors, also modeled by means of a graph. The pattern of connections is represented by an undirected graph $M=(P, L)$. Vertices represent processors and if two vertices a and b, correspond to two directly linked processors, then an edge (a, b) is included in the edge set of the graph.

The most important feature here is the *distance* between two processors a and b. It is realistic, especially for VLSI circuits, to make the following simple assumptions:

1. The distance between two connected processors is of unit length.

2. All processors are capable of processing every possible task.

3. The processing speed is the same for all processors and all tasks.

4. If processor a can communicate with processor b then also b can communicate with a.

We thus obtain an undirected connected graph $M=(P, L)$, where for $a, b \in P$ we are interested in a distance function $\Delta(a, b)$ defined as follows:

- If $(a, b) \in L$ then $\Delta(a, b)=1$.

- Otherwise, $\Delta(a, b)$ is equal to the length of the shortest path from a to b.

(However, the restriction to edges of unit length is not indispensable. If edges are allowed to have arbitrary length, then $\Delta(a, b)$ can be defined as the length of the shortest path from a to b.)

To determine how the computation will be carried out, we first need a *parallelization* function $\Pi: V \to P$, which denotes that a task $\alpha \in V$ will be processed by processor $\Pi(\alpha)$. We also need a *timing* function $X: V \to \mathcal{N}$, which denotes that the execution of a task $\alpha \in V$ will start at time $X(\alpha)$.

Two types of relationship must be respected:

- the *precedence relationship*, which is expressed by the edges in G. If the duration of a task α is $\tau(\alpha)$ time units and its execution started at time instant $X(\alpha)$, then the execution of a task β which depends upon $\alpha(\alpha \to \beta)$, may start only after α has finished, namely $X(\beta) \geq X(\alpha)+\tau(\alpha)$.

- the *concurrency relationship*. Each processor can deal with only one task at a time. Hence, if two tasks are executed simultaneously, they must be processed by different processors. Thus, if

$$[X(\alpha), X(\alpha)+\tau(\alpha)] \cap [X(\beta), X(\beta)+\tau(\beta)] \neq \varnothing, \text{ then } \Pi(\alpha) \neq \Pi(\beta).$$

We now have a model representing the idea that "a computation is performed in parallel by a machine". There are many possible ways to carry out a computation. Our problem is how to find and evaluate these possibilities. What we are actually expecting is to perform a computation within a tolerable cost range, where the major components of the cost are the *computation time cost* and the *cost of interprocessor communication*.

The *time cost* is the interval from the moment the first task started execution to the moment the last task finished its execution:

$$T = \max_{\alpha}(Y(\alpha)) - \min_{\alpha}(X(\alpha)) \tag{1}$$

where, for every task α, we define the function $Y(\alpha) = X(\alpha) + \tau(\alpha)$.

The *communication cost* is the sum of all distances $\Delta(a, b)$, such that the distance $\Delta(a, b)$ is taken into account each time processors a and b communicate with each other. Two processors communicate if one of them transmits data to the other, *i.e.*, if the task processed by one is prerequisite to the task processed by the other. Obviously, this situation corresponds to an edge of the graph G. Hence, the communication cost will be given by:

$$C = \sum_{(\alpha, \beta) \in E} \Delta(\Pi(\alpha), \Pi(\beta)) \tag{2}$$

We are now in a position to formulate a model for the execution of computations on a network of processors.

Given

- a computation $G=(V, E)$, together with a duration function $\tau: V \to \mathcal{N}$,
- a machine $M=(P, L)$, together with a distance function $\Delta: P \times P \to \mathcal{N}$,
- cost bounds T_m, C_m for time and communication,

find

- a timing function $X: V \to \mathcal{N}$,
- a parallelization function $\Pi: V \to P$,

such that $T \leqq T_m$ and $C \leqq C_m$.

The range of tolerance for the costs can be described in various ways. In early work done around 1970 [6, 7], the bound C_m was in fact equal to

infinity, since the main interest focused on computation time. Other approaches consider bounds on the total cost, *i.e.*, that $T+C \leqq S_m$ [1]. Moreover, a practical limitation is the usual assumption that all processors can directly communicate with each other, *i.e.*, that $\Delta(a, b)=1$, for all a, b, (which means that our machine $M=(P, L)$ is a *clique* [3, 11]).

During the last years, there is an increasing effort in analyzing machines where processors are not fully connected. There is something common in all these approaches: they try to reduce the cost (T, C) in a single step. This is due to the fact that minimization of computing time implies increased parallelism (hence, communication cost), whereas minimization of communication means using a single processor, thus leading to a trivial problem. We introduce here the idea of applying an *ad hoc* but reasonable condition of parallelism and of scheduling the computation in two steps: first find a non trivial parallelization Π and then a timing X. The first results in this direction are presented in the following section.

2. MAIN RESULTS

Consider a computation graph $G=(V, E)$ and two tasks α and β such that $\alpha > < \beta$, *i.e.*, none of the two tasks is prerequisite to the other. Then, α and β are parallelizable, in the sense that they could be processed simultaneously. We can define, therefore, a *parallelizability* relation *par* (α, β) and ask for a parallelization that respects this relation and assigns parallelizable tasks to different processors. Given such a parallelization, we may hope that a good timing can be found, since tasks that could be processed in parallel have already been assigned to different processors and, in principle, can be processed simultaneously. Our aim is to provide a suitable relation *par* (α, β), such that a parallelization can be obtained in polynomial time.

Let $G'=(V, E')$ denote the computation graph constructed from G by removing all transitive edges, *i.e.*, all edges (x, y) such that there exists a path from x to y. We define *par* (α, β) to hold between two tasks whenever α, β are siblings in this graph, *i.e.*, if and only if there is a task $\gamma \in V$, such that $(\gamma, \alpha) \in E'$ and $(\gamma, \beta) \in E'$. This does not mean that these tasks will always be executable in parallel immediately after their father has been executed, but it is considered a "safe" policy to attach them to different processors. Let H denote the largest subset of V, such that if $\alpha, \beta \in H$ then *par* (α, β). We will suppose for simplicity that $|H| \leqq |P|$, *i.e.*, that there will always be enough processors to process simultaneously all parallelizable tasks of the graph.

We will be interested in finding a parallelization Π that respects the relation *par*, *i.e.*, such that $par(\alpha, \beta) \Rightarrow \Pi(\alpha) \neq \Pi(\beta)$. We thus modify the original problem keeping the same data but asking for:

- a parallelization Π respecting the relation *par* and having cost $C \leqq C_m$,
- a timing X respecting Π and having cost $T \leqq T_m$.

In what follows, we investigate the first problem.

PROPOSITION 1: *Given a computation graph* $G=(V, E)$ *and a machine* $M=(P, L)$, *then finding a parallelization* Π *that respects the relation* par *and has cost less than or equal to* C_m, *is an NP-complete problem.*

Proof: The proof comes from a reduction of the *Optimal Linear Arrangement problem* (*OA*) and a restriction of our problem to a machine M consisting of a linear array of processors. *OA* is an *NP*-complete problem ([8], p. 200). Our problem obviously belongs to *NP*: it can be verified in polynomial time that a given parallelization respects *par* and its cost can be computed. By the reduction given below, it follows that our problem is also *NP*-complete.

An instance of the *OA* problem consists of a graph $A=(V_a, E_a)$ and a number K_{OA}, and asks whether there exists a mapping $f: V_a \rightarrow (1, \ldots, |V_a|)$ such that the cost-sum D of $abs(f(u)-f(v))$ over all edges (u, v) of A is less than K_{OA}. For each instance of *OA* we construct an instance of our problem, with an array of processors p_j, $j=1, \ldots, |V_a|$, and a computation graph G defined as follows. (i) For each node u_j of A we define a corresponding node U_j of G. (ii) For each edge (u_i, u_j) of A we define a corresponding node E_{ij} of G. (iii) We define a special node R of G. (iv) We define in G all edges (E_{ij}, U_i), (E_{ij}, U_j), and all edges (R, U_i). Now, by the definitions above, $par(U_i, U_j)$ holds for all i, j, so U_i, U_j must be assigned to different processors, whereas E_{ij}, R are free to be assigned to any processor. We can prove that the graph A can be arranged with cost D less than K_{OA} if and only if the so constructed computation graph G can be assigned to the machine M with communication cost C, less than $C_m = K_{OA} + C_R$, where C_R is a constant that will be defined below.

If Part. As $par(U_i, U_j)$ holds for all i, j, in order to obtain a minimum communication assignment, the tasks U_j, $j=1, \ldots, |V_a|$, must be assigned to the $|V_a|$ processors one to one; consider that U_j is assigned to $p_{f(u_j)}$. Let C_E denote the communication cost corresponding to the nodes E_{ij}. Each task E_{ij} is free to be assigned to any processor of the array in the range from $p_{f(u_i)}$ to $p_{f(u_j)}$, thus contributing to the communication cost C_E by the quantity $abs(f(u_i)-f(u_j))$. The node R is free to be assigned to the "middle" processor

minimizing its communication cost to the U_j nodes to the value C_R. (If $|V_a| = 2l+1$ then C_R will be equal to $l(l+1)$; if $|V_a|$ is even then C_R will be defined in an analogous manner.) We keep the same f for assigning nodes u_j of A to positions $1, \ldots, |V_a|$. The arrangement cost D will be equal to C_E, while the communication cost C is equal to $C_E + C_R$. By the above definitions, D will be less than K_{OA} if C is less than $C_m = K_{OA} + C_R$.

Only If Part. Let the nodes of A be arranged by f, with optimal cost D less than K_{OA}. We assign each U_j to processor $p_{f(u_j)}$, each E_{ij} to any processor in the range from $p_{f(u_i)}$ to $p_{f(u_j)}$, and R to the middle processor. As before, we will have $C = D + C_R$, so D can be less than $K_{OA} = C_m - C_R$ only if C is less than C_m. ■

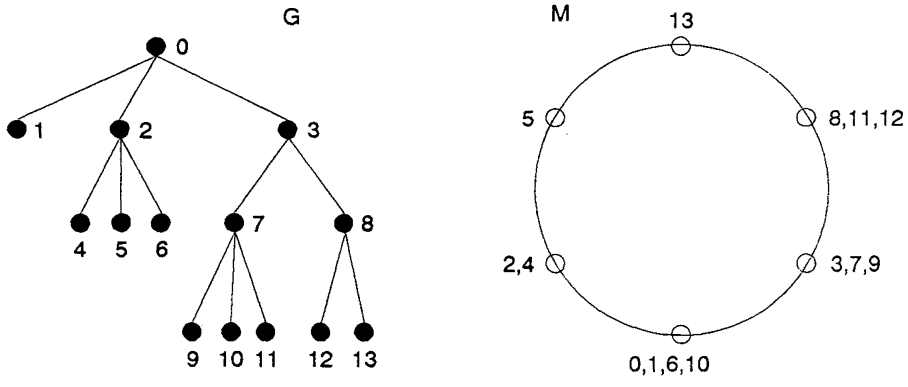

Figure 1. – A tree computation assigned to a ring graph.

There are, however, cases of parallelization problems that can be solved in polynomial time. If the computation graph G is restricted to be a tree, the problem can be solved in polynomial time for a wide class of machine graphs. This case will be the subject of the remaining of this section. We will first discuss a fundamental example, and, in the sequel, will prove that a whole subclass of the parallelization problem, which is of great practical importance, can be solved in polynomial time.

PROPOSITION 2: *The problem of the optimal parallelization for a computation G and a machine M can be solved in polynomial time, if the machine is a* ring *and the computation is a* tree.

Proof: We assign the descendants of a node symmetrically around it at least distances (*i.e.*, one child is assigned to the same processor as the parent node

and the remaining children on the left and right of that processor in a balanced manner) and proceed recursively upwards (see *fig.* 1). Since trees can be "shifted" along the ring without changing the communication cost, the whole assignment will be optimal. ■

The key argument that, if we have the optimal parallelization for a subtree, then we can "move" this subtree on the ring of processors without disturbing the optimality of its parallelization, is meaningful, since the ring is a very simple graph (there is a direction and only one). We will now extend this notion of "shifting" on general graph structures.

Consider a graph $M=(P, L)$ with distances between its nodes, as defined in the introduction, the distance between two adjacent nodes not being necessarily equal to 1. A *consistent displacement* on this graph is a bijection of the set of vertices to itself, $\Phi: P \to P$, that preserves the distances between nodes, *i.e.*, $\Delta(a, b)=\Delta(\Phi(a), \Phi(b))$, $a, b \in P$. (Obviously, since Φ is a one-to-one mapping, it suffices that distances between adjacent nodes be preserved.) Note that if the distance between two adjacent nodes is set to one, then "consistent displacement" is a notion equivalent to "isomorphism". We will be interested in *step* displacements, namely in those consistent displacements where there is a node p that is shifted to a node adjacent to it, *i.e.*, $(p, \Phi(p)) \in L$.

Finally, an *affine graph* is defined as a connected graph, with a length defined on each one of its edges, such that for each node p and every node q adjacent to p there exists a step displacement which maps p to q, *i.e.*, $\Phi(p)=q$. In the case where the distance between two adjacent nodes is equal to one, this notion reduces to the one of *point symmetric* graphs [10] or *vertex transitive* graphs [4, 13]. If we assign a tree to an affine graph, then its root and the whole tree subsequently, can be shifted by one position in an obvious manner, namely, by considering the node p of M corresponding to the root of the tree and a step displacement Φ mapping p to one of its adjacent nodes. Each node of the tree is then mapped to a new node of the graph following the displacement Φ.

The next proposition is the natural extension of the last one and states that the optimal parallelization problem for a tree can have a very efficient solution in the case of affine graphs, namely of a large class of graphs. In fact, affine graphs provide a generalization of point symmetric graphs by allowing the definition of appropriate valuation on the edges. Thus, beyond rings, doubly connected grids (*torus*) and infinite plane grids are affine graphs, even if distances between adjacent nodes are different on different directions. Also, regular polyhedra are affine, whereas, generally, bipartite graphs or

grids are not. The practical relevance of these considerations is enhanced by the fact that there exist machine implementations of doubly connected grids, as well as of bipartite graphs.

PROPOSITION 3: *The problem of the optimal parallelization for a computation G and a machine M can be solved in polynomial time, if the machine is an* affine *graph and the computation is a* tree.

Proof: Consider a task tree $G=(V, E)$ and a machine $M=(P, L)$ whose connections are as described above. Let W be a subtree of G. The cost of the parallelization $\Pi: V \to P$ is the sum of the distances $\Delta(\Pi(\alpha), \Pi(\beta))$ over all the edges (α, β) of the graph G. The set of edges (α, β) can be partitioned into three categories: (1) α, β in W, (2) α in W, β not in W, and (3) α, β not in W. Case (2) implies that α is the root of the subtree.

Let us now consider a mapping Φ from M to itself producing a consistent displacement, as discussed before, and let us "shift" only the tasks in W, thus defining a parallelization Π', such that $\Pi'(\alpha)=\Pi(\alpha)$, if α not in W, and $\Pi'(\alpha)=\Phi(\Pi(\alpha))$, if α in W. We can observe the following in relation to the three cases considered above. The distances are not modified in case (1), and tasks satisfying the *par* relation are still assigned to different processors (Φ is a one-to-one mapping). The distances are modified in case (2), as the connection of the root of W is modified. There is no concern about *par* (α, β) in that case. Finally, the distances are not modified in case (3), and tasks satisfying the *par* relation are still assigned to different processors.

Hence, Π' is a new parallelization of the tasks and the only alteration it causes to the cost with respect to Π is due to change of distance on the connection of the root of W with its parent node. Suppose that this root is assigned to processor p_0 and that, if it was assigned to processor p_k, the cost of the connection with its parent node would be less. There is a path in M from p_0 to p_k and since M is an affine graph, it is possible to find for each pair of successive nodes on this path a displacement mapping one node to the other. If Φ is taken as the composition of the above displacements, we can shift the subtree W keeping all communication costs constant except for the cost concerning its root, which is decreased, thus decreasing the total cost.

Consider now a processor p and let $P_1, P_2, \ldots$ be the sets of processors at distances 1, 2, ... from it respectively. If p processes the task ρ, then the least communication cost is achieved by assigning one descendant of ρ to processor p, $|P_1|$ of the descendants of ρ to the corresponding processors of

the 1st level, $|P_2|$ to the 2nd level, and so forth, proceeding at least distances from p.

We assign the descendants of a node around it, as explained above, and proceed recursively upwards. Since trees can be repositioned by "shifting" on the graph without changing the communication cost, the whole assignment will also be optimal. The problem of searching for the nearest nodes can be solved in polynomial time by breadth-first search or, more generally, by applying Dijkstra's algorithm [2]. ■

We should be able to test whether a graph is affine or not. Because of the involvement of the notion of isomorphism, this problem can be easily reduced to the graph isomorphism problem in polynomial time. There is evidence [5], that the graph isomorphism problem is not *NP*-complete, but an algorithm for it in *P* has not yet been found. However, there exist polynomial algorithms for the graph isomorphism problem (and consequently for our problem), if the problem is restricted to degree bounded graphs [12]. Since the graphs considered here represent interconnections of processors, this constraint is not always so restrictive.

It can also be mentionned that there exists a technique based on group theory, which produces all point symmetric graphs ([4], p. 108, 111, [13]).

We shall not elaborate this subject further, because we have no conclusive evidence that affine graphs represent all that we should be looking for. It is possible that Proposition 3 holds also for other – wider – classes of graphs.

3. TIME CONSIDERATIONS AND CONCLUSION

The above introduced two-phase method tries to minimize the communication cost and, in the course of doing so, seems to assign too many tasks to neighbouring processors during the first phase, thus increasing the lower bound of the timing searched for during the second phase. Nevertheless, let us consider as an example a set of tasks arranged as a full tree of out-degree d and height h to be executed on a simple machine consisting of a long enough line of processors. This case has some bad features: it is a shallow tree which makes the parallelization algorithm overschedule the processor executing the root task. No sequential execution of these $N=(d^{h+1}-1)/(d-1)$ tasks can take less than N units of time. Their parallelization, done as explained above, assigns the root of the tree to some processor. Let $n_{i,j}$ denote the number of tasks of the i-th level of the tree ($0 \leqq i \leqq h$) assigned to a processor at distance j from the "central" processor executing

the root task. Suppose $d=2r+1$ for easiness. By the recursive way of assigning tasks, we have that $n_{i,j}$ is equal to the sum of $n_{(i-1),k}$ for $k=j-r, \ldots, j+r$. With the root at $i=0$, one can verify that $n_{i,j}=O(d^{i-1})$ for $i \geqq 1$. The number T of tasks assigned to the central processor is given by the sum of $n_{i,0}$ for $i=0, \ldots, h$ which is $O(d^{h-1})$. This is an exponential number of tasks (with respect to d), but we already had an exponential number of tasks, namely $N=O(d^h)$. However, we used only $W=O(h*d)$ processors. Therefore, the product $T*W$ is $O(N \log N)$, so this parallelization is not optimal but only for a $\log N$ factor. (In fact the factor is slightly less. Numerical results suggest a $\log N/\log\log N$ factor.) Can we achieve more? We suggest to redefine *par* to include pairs of nodes having a common ancestor situated two levels above them (instead of one), thus obtaining d^2 instead of d (and proceed in a similar manner for higher degree acceleration). Since the Optimal Linear Arrangement is polynomially solvable for trees, we do have some hope that a parallelization can be found in polynomial time achieving greater acceleration. We need more on this point, since a knowledge of the structure of the neighbourhoods of nodes on affine or point symmetric graphs seems necessary.

The problem of finding an optimal parallelization for a given algorithm is complemented by the problem of finding an optimal timing that respects the parallelization. The latter is a well known problem in computer science (precedence constrained scheduling) [6]. Even simple instances of this problem are *NP*-complete and, hence, impractical to solve. There is, however, a case (and may well be more) that fits to our model: if the partial ordering of the tasks is a cyclic collection of trees, then the problem of the optimal timing can be solved in polynomial time [9]. This means that, on a large category of machines, a class of problems accepts an optimal parallelization and, in the sequel, an optimal timing that can be determined in polynomial time.

In our case, we can look forward to something more: the parallelization obtained is not arbitrary, but respects the relation *par*. It might be possible to take advantage of this fact, perhaps varying *par* itself, in order to enlarge the class of problems ("machines" and "computations"), for which the optimal timing can be found in polynomial time.

ACKNOWLEDGEMENTS

The authors wish to thank the anonymous referee for carefully reading the manuscript and for his/her useful comments and suggestions.

REFERENCES

1. F. AFRATI, Ch. PAPADIMITRIOU and G. PAPAGEORGIOU, *Scheduling Dags to Minimize Time and Communication*, Aegian Workshop on Computing (AWOC), Corfu, June 1988.
2. A. V. AHO, J. E. HOPCROFT and J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
3. S. G. AKL, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, 1989.
4. N. BIGGS, *Algebraic Graph Theory*, Cambridge University Press, 1974.
5. R. B. BOPPANA, J. HASTAD and S. ZACHOS, Does Co-NP Have Short Interactive Proofs?, *Information Processing Letters*, Vol. 25, 1987, pp. 27-32.
6. E. G. COFFMAN and P. J. DENNING, *Operating Systems Theory*, Prentice-Hall, 1973.
7. E. G. COFFMAN (Editor), *Computer and Job-shop Scheduling Theory*, John Wiley, 1976.
8. M. R. GAREY and D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., San Francisco, 1979.
9. D. K. GOYAL, *Scheduling Processor Bound Systems*, Report No CS-76-036, Washington State University, 1976.
10. F. HARARY, *Graph Theory*, Addison-Wesley, 1972.
11. R. M. KARP, *A Survey of Parallel Algorithms for Shared-Memory Machines*, Report No. UCB/CSD 88/408, University of California Berkeley, March 1988.
12. M. E. LUKS, Isomorphism of Graphs of Bounded Valence Can Be Tested In Polynomial Time, *Journal of Computer and System Sciences*, Vol. 25, 1982, pp. 42-65.
13. G. SABIDUSSI, Vertex Transitive Graphs, *Monatshefte für Mathematik*, 68, 1964, pp. 426-438.