SCHEDULING OF TASKS FOR DISTRIBUTED PROCESSORS

by

R. Mehrotra and S.N. Talukdar

DRC-18-68-84

December, 1984

SCHEDULING OF TASKS FOR DISTRIBUTED PROCESSORS

Ravi Mehrotra
Electrical and Computer Engineering Department
N.C. State University, Raleigh, North Carolina 27650

Sarosh N. Talukdar
Department of Electrical Engineering
Carnegie-Mellon University, Pittsburgh, PA 15213

ABSTRACT

The paper describes a technique for estimating the minimum execution time of an algorithm or a mix of algorithms on a distributed processing system. Bottlenecks that would have to be removed to further reduce the execution time are identified. The main applications are for the high level design of special purpose distributed processing systems.

The distributed systems are modelled by P, a set of nonidentical processors and R, a set of resources that the processors can use. The algorithms are modelled by T, an ordered set of tasks. The problem of optimally assigning the processors to the tasks while meeting the resource constraints is NP-complete. However, a heuristic using maximum weighted matchings on graphs has been devised that is extremely fast and comes reasonably close to the optimal solutions.

1. INTRODUCTION

Our main concern in this paper is assignment of the tasks of an algorithm or a mix of algorithms for distributed processing - not the distributed processors themselves. We take a fairly high level view of distributed processors. Specifically, we will think of a distributed processor as consisting of two sets - one of processors and another of resources that the processors can use. The collaboration amongst processors can be intimate - processors may access one another's memories - and use fast communication networks so that arrangements that one traditionally called multiprocessors are also included in our view of distributed processing. We will deal with the problem of assembling a distributed processor from a given mix of components, to compute a given synchronous algorithm, or mix of algorithms, in minimum time. The relevance of this problem is explained below.

If an algorithm has a great deal of regularity and a great deal of fine grained parallelism (at the instruction level), then it is best to vectorize it and use pipeline or array processors for its execution. Distributed processing, because of the relatively large communication overheads it entails, is better suited to exploiting coarser grains of parallelism, such as occur among large blocks of instructions or among major tasks. At this grain, regularity is less common. It is unusual for an algorithm's major tasks to be identical or even similar. More often an algorithm contains a mix of quite different tasks with quite different processing requirements. One consequence is that homogeneous machines (i.e., machines with identical processors that are symmetrically connected) can, at best, be designed so that their processors are compatible with the average task. Bottlenecks invariably develop in the processing of non-average tasks. Of course, there is no reason to restrict distributed processors to homogeneous structures. There is a very wide variety of available processing elements ranging from large, general purpose main frames like the CRAY-1 to VLSI chips that are dedicated to a single function. This variety makes for a very large number of alternate structures for distributed processors. How is one to find a good alternative for a given application? A high level approach to answering this question is obtained by representing algorithms by graphs (as described in Section 2.1) and thinking of a distributed processor as consisting of two sets - one of processing elements, another of resources (such as I/O devices and interconnecting devices) for the processing elements to use. To continue the approach one may take the following steps:

1. Select a graph representation of the algorithm (or mix of algorithms) under consideration.
2. Formulate the constraints governing the processor set, the resource set and their interactions.
3. Estimate the time that each processor will take to execute each node (task) of the graph.
4. Find the optimum assignment (that is, find the subsets of processor and resources and find the allocation of tasks to processors that minimizes the execution time of the algorithm while meeting the constraints of step 2).

The result of completing these four steps will be the high level design of a nonhomogeneous distributed processor.

In this paper, we will very briefly discuss the first three steps and then devote the bulk of our attention to the fourth and most difficult step.

The first step involves tearing the algorithm into tasks and identifying the ordering constraints on them. The parent graphs can be torn into more elaborate offsprings but not till the tasks become *very* small (approaching individual instructions) do the graphs display enough structural variety to warrant systematic procedures for their generation. Of course, when the tasks are this small, distributed processing is much less desirable than array or pipeline processing. In summary, we feel that the first step is best done by inspection.

The second step involves selecting the set of processing elements to be considered for inclusion in the distributed processor and articulating any other relevant constraints, such as a limit on the total cost of the distributed processor.

Much of the information called for In the third step is available in tfie literature. For example, one can readily find experimental data on the times taken by various array processors to complete L-U factorizations of matrices which have the sparse structures that occur in power systems. When the requisite information is unavailable in published or manufacturers[1] literature, benchmarking or detailed simulation may be undertaken to obtain it.

The fourth step - finding an optimum assignment - is a *very* difficult constrained minimization problem. The remainder of this section will be devoted to developing and illustrating a heuristic for solving it. The heuristic *is* efficient and finds solutions that are reasonably close to optimal solutions.

The rest of this paper is organized as follows. Section 2 gives a formal description of the minimization problem. Section 3 briefly reviews available methods for tackling similar problems and lists their principal failings. Then Section 3 goes on to develop a heuristic which translates the •minimization problem into one of Maximum Weighted Matchings (MWM). The procedures of Section 3 have been coded to give a friendly, FORTRAN program called SNONUET. The usage and features of SNONUET are illustrated in Section 4.

2 MATHEMATICAL FORMULATION OF THE PROBLEM

This section establishes the basic vocabulary for the remainder of the paper and gives a precise mathematical formulation of the problem to be considered.

2.1 Algorithms

Recall that an algorithm A is described by A = (T,a}, where T is a set of tasks *il*$_j$, T_2,... T_N> and the set a denotes the partial ordering relation on T such that T_p a T_s implies that the execution of tasks T_s (called the successor of T_p) cannot begin until the execution of T_p (called the predecessor of T_s) has been completed. We will represent an algorithm A by a directed graph ! called the Task Order Graph $G_A(V,E)$ of A so that there is one node in V for each task in T and one arc in E for each relation in partial order a. When a is empty the tasks are called independent.

It is assumed that the tasks to describe A are chosen from a finite set of tasks T^P = {T^, $T_2{}^P$, • .. $T_n{}^P$}ofn primitive tasks. Each task T_x?T corresponds to some primitive task $T_y{}^P$£T^P.

2.2 Distributed Computers

At a high level, we may think of distributed computer architectures as assemblies of processors that can execute tasks in TP provided that they have access to certain resources such as disk drives, tapes, memory and interconnecting devices such as buses and data links. We will represent a distributed computer system with M processors and L resources by MP{P,R} where P = {P], P2.... PM> is a set of the M processors and R ^{s} {R], R£, ...RL> is a set of the L resources.

2.3 Algorithm - Distributed Computer Interactions

Each task T_i£T may be executed on any processor in P. We define a function n(T_i.) * (t_{ir} t_{21},...t_{M1}) so that the value of t_{ri} represents the expected time it takes to execute task T_i on processor P_r. Furthermore, we define a function r(T_i) * (^{r}ii» ^{r}2i•'"'^{r}1i^ *o represent tne resource requirements of task T_i such" that r_{xy} is equal to the arrount of discrete resource R_x needed while executing T_y and 8(R_x) is the total units of R_x in the system.

2.4 Execution Time of A on MP(P,R}

Let T(T.) represent the starting time of the execution ofUask T^T.

Define X(kl = 1, if task T_j is executed on processor *P* in time interval k and zero otherwise. It is assumed that time is measured in terv.s of equal and indivisible units. Using the notation introduced in this section we define a feasible schedule to be a mapping *$: T+]* where 1 is a one dimensional space of integers representing time, such that the following 3 conditions are satisfied:

$$\sum_{1-1}^{m} X_{iJ}(k) = 1 \quad \text{for } j=l..N, \text{ all kei} \tag{1}$$

If T_i aT_j then

$$T(T_J) > T(T_]) + \sum_{H}^{m} t_{rl} X_{rl}(k) \quad \text{fori.j-l-.N, all kel} \tag{2}$$

$$6(R_1) > \sum_{">1}^{N} \sum_{p-1}^{m} r_{1J} X_{PJ}(k) \quad \text{for all kel} \tag{3}$$

Eq. 1 is needed to avoid the assignment of a task to more than one processor. Eq. 2 is a statement of the procedure constraints of A. Eq. 3 is needed to ensure that the resources required by a job will be available while the job executes.

Corresponding to each feasible schedule *in* T-*l we define the execution time of the algorithm A on MP as:

L^M£ « min {X_{ij}(k)=O for 1*1..m, J-1..N | k > 0}

Thus, the problem of finding the optimal assignment of tasks in the algorithm A on a distributed computer MP, so as to minimize the overall execution time, may 5e stated as:

GSP:

Minimize L_A

subject to t r M (4)

For every node $n.\hat{}cG_A(Y,E)$ define the weight of n_1, W C ^), as *U(n^)* * min (t^, t_{21},. ., t_{mt}). Define the length of a directed path from node n_s to node n_t to be equal to the sum of weights of all the nodes in the directed path from n_s to *n+*. The longest directed path from a node with no predecessor to a node with no successor represents a lower bound on L^. This lower bound L^ is obtained by assuming that all the tasks in $G_A(V,E)$ are assigned to the processor on which they take mir.iir.um execution time and there is a sufficient number of processors and resources in the system.

Ifpthe solution to GSP of Eq. 4 gives a value of L^{H}_{A} > *l%** the eleiıents of the set p dnd R mftX be modified to reduce the difference between *v*\ and LJ. This allows us to reconfigure a distributed tcr.puter system to rake it best suited for executing the algorithm under consideration. The solution procedure (Section 3.1) used to solve GSP enables us to identify the elements of MP that limit performance, thus suggesting a natural modification of the set P and/or R.

2.5 A Cost Constraint

Let CP(Pi) represent the cost of processor Pj, i^sl...M. A cost constraint may be added to GSP as follows:

GSPC:

ftinimize L^*_A

subject to ψ:T→1

$$\sum_{i=1}^{M} CP(P_i)\, Y_i \pm COST \qquad (5)$$

All processors in the set P are not necessarily used. The solution procedure attempts to select the particular mix of processors that minimizes the overall execution time with the total cost of processors in the mix being no more than COST. Note that Y-J is 1 if processor Pi is assigned some task and 0 otherwise.

3 SOLUTION PROCEDURE

GPS is a well known and notoriously hard problem (the name is an acronym for General Scheduling Problem). Much of the work in the area has been devoted to the subproblem of GSP in which all the processors are identical [1, 2, 3, 4, 5, 6, 7, 8, 9]. Some work has also been devoted to the use of enumerative and iterative techniques, such as local search and branch and bound, for other subproblems of GSP [10, 11, 12, 13]. However, we know of no techniques that adequately address the more general version of GSP that is of interest here,

It happens that GSP is NP-complete.[*1] This means that it is unlikely that an algorithm can be devised to find optimal solutions of GSP in any reasonable length of time. A better strategy is to seek heuristics that will find reasonably good solutions in reasonable amounts of time. We will proceed to develop one such heuristic.

3.1 A Heuristic for Solving GSP

The heuristic technique is based on finding maximum weighted matchings on graphs. Its essential steps are:

1. Determine the Edge List Matrix of $G_A(V,E)$, the Execution Time fetrix [t^] and the Resource Requirement Katrix [r^] ^{J}(see the Example of Section 3.3.3).
2. Assign levels to the nodes, T_x, of the task graph G.(V,E). Intuitively, the level of a node is^its distance from a node with no successor or a nc-^e with no predecessor. As such, levels represent the precedence structure of G_A(V.E).
3. Making use of the levels of the nodes, assign tasks to the processors while disregarding the resource constraints. This step is carried out by finding maximum weighted matchings on graphs.
4. Schedule the tasks on the processors they have been assigned to, taking resource constraints into account, f'ake a list of resource shortages if any.
5. Repeat steps 3 and 4 until all tasks have been scheduled.
6. Output the schedule and the list of resource shortages (see example of Section 3.3).

We will now proceed to describe how steps 2-5 may be taken.

The nodes, T_x of the task graph $G_A(V,E)$ of an algorithm A are assigned two levels $L^R(T^x)$ and $L^F(T^x)$ by the following algorithm:

Assign-Levels:

1. If T^x has no successors, then $L^B(T^x) = 1$; otherwise, $L^B(T_x) = 1 + \max\{Lg(T_x) \mid T_x a T_y\}$.
2. Let $L^B(T_{max})$ represent the smallest integer such that $L^B(T_{max}) \geq L^B(T_x)$ for all tasks T_x.
3. If T_x has no predecessor, then $L^F(T_x) = L^B(T_{max})$; otherwise, $L^F(T_A) = \min\{L^F(T_J) \mid T_J\ a\ T_*\} - 1$.

[*1] This is a class of difficult problems. Known algorithns for finding their optimal solutions require execution times that increase exponentially with problem size [6].

The tasks T_xeT are *first* assigned to processors PycP without regard to the resource constraints of R_2eR (but taking the precedence constraints into account) by the algorithm assign-tasks and then scheduled on the corresponding processors by the algorithm schedule-tasks. The tasks in T are scheduled in the decreasing order of their levels L^B(T), taking resource constraints into account. If resource constraints are violated, the starting time of the task is delayed until a sufficient amount of resource is released *by* tasks which complete execution. While scheduling tasks, the algorithm, schedule-tasks, ensures that the conditions of Theorem 1 are satisfied. In order to understand how the scheduling procedure works, it is convenient to assure that all tasks T_y with $L^B(T_y)$ > 1 have already been assigned to processors and scheduled on then. Consider the set % of tasks T_x such that $L^D(T_x)$ = 1. Let • {$T_{x,t}$ T_x , . . . T_{vl} .} and define the set J = {1,2,..!|-H}^{2}so that the elements of J are in one to one correspondence with tasks in *. Let PI • {l,2...m> represent the set of processor indices to which tasks are to be assigned without regard to the resource requirements. This assignment problem may be formulated as an N?-complete Integer Linear Program as follows:

ILP:

Minimize COMP TIME

Subject to $\sum_{j \in J} t_{ij}\, z_{ij}$ - COMP TIME $\leq$ 0 all iePl

$\sum_{i \in Pl} z_{ij}$ « 1 all jeJ

z_{ij} s 0 or 1 all iePl & jeJ (6)

The solution to the above ILP gives the optical processor assignment that minimizes the latest finish tiire, COMP TIME, of the independent task set *' ^{2}ii^ ^ ^{T}x-$_j$ $^{1S\ assi*S^{ned\ t0}}$ processor P. and z_{ij} s 0, otherwise. ILP can be solved by a general ILP algorithm such as a cutting-plane method or branch and bound but such solution procedures are NP-coiTiplete. Instead, we solve ILP by transforming it to another problem given below:

ILPM:

Maximize $\sum_{i \in Pl} \sum_{j \in J} c_{ij}\, y_{ij}$

Subject to $\sum_{j \in J} y_{ij} < b_i$ all iePl

$\sum_{i \in Pl} y_{ij}$ • 1 all jeJ

y_r « 0 or 1 all iePl & all jeJ (7)

The y_{ij}'s have the same interpretation as the Zjj*s

The Cjj's and b*'s are defined by the altorithm assign-tasks. ILPM is known as the Maximum Weighted Matching problem (MWM) [14, 17, 20]. A polynomial time algorithm to solve MWM on bipartite graphs is described in [15, 16]. Solution to ILPM yields an upper bound, UB, for the solution to ILP. The inequality constraints and the objective function of ILPM are modified to improve UB and bring it closer to the solution of ILP.

Assign-Tasks:

1. Initialize as follows:
   a. b_i = |J|, for all iePl.
   b. c_{ij} * {$\sum_{i \in Pl} t_{ij}$> / t_{ij} for all iePL & jeJ
   c. UB = • and MIX » FALSE
2. Solve ILPM (MM) to determine y^ for uPl, jeJ
3. Evaluate tp^ and i^* as follows:
   a. tp_i = $\sum_{j \in J} t_{ij}\, y_{ij}$, forall iePl
   b. 1* = {x| $tp_x \geq tp_i$ for all iePl}
   c. If CMTC1 is TRUE) go to step 6.
   d. if (max {tp^ iePl ^ UB) go to step 5.
4. Evaluate b_i, b_i* and z_{ij}, as follows:
   a. b_i = |J| for all iePl
   b. b_i^* = ($\sum_{j \in J} y_{i^*j}$) - 1
   c. z_{ij} « y_{ij} for all iePl, jeJ
   d. UB = tp_{i^*} & Go to step 2.
5. Update b_i, b^ and c_{ij}, as follows:
   a. b_i « |J| iePl
   b. 5_i* « ($\sum_{j \in J}$ y$_v$) - 1
   c. MTC = TRUE
   d. C_{ij} « { $\sum_{i \in PL} t_{ij}$ / $\sum_{j \in J} t_{ij} Z_{ij}$ /t_{ij}, all iePl &jeJ
   e. Go to step 2.
6. Check for further improvement and update z_{ij}: as follows:
   a. If (tp^ $\geq$ UB) go to step 7.
   b. z^ = y^ for all iePl, jeJ
   c. UB_f = tp_i*
   d. Go to step 5.
7. Evaluate the final values for tp_i, b_i and COMP TIME as follows:
   a. tp_i « $\sum_{j \in J} t_{ij}\, z_{ij}$ for all iePl
   b. b_i • $\sum_{j \in J} z_{ij}$ for all iePl
   c. COMP TIME = tp_{i^*}

Assume that all tasks T_x, such that $L^B(T_X)$ • 1, have been assigned to processors using the algorithm assign-tasks. Before scheduling the as-signed tasks to processors, we form the following 2 sets:

I. If Zjj (see step 4c & 7a of the algorithm assign-task) is 1, task T_{x_j} has Been assigned to processor P^. For each jePl, form a set Jj « {j | Zjj * 1} and a set ^ * (T_{x_j} | z^ * 1,

$\sum_{k=1} r_{kx_i} \leq \sum_{k=1} r_{kx_{i+1}}$

for i= 1..| -1). Set ^ is a list of tasks that have been assigned to processor P* in increasing order of their total resource requirements.

2. For each set J_i, iePl, tp.j (see step 3a & 7a of the algorithm assign-task} represents the total time taken on processor P_i assuming all tasks assigned to it could be executed on it in succession without violation of resource constraints. Form a set TP * (tp_i f 0, for all iePl | tp_1 £ tp_{t+1}>. Note that the set TP may have fewer than n elements.

The assigned tasks are scheduled as follows. The processors are considered in the increasing order of tp_i (see step 3a & 7a of the algorithm assign-task) and tasks assigned to them are scheduled on the processors. Thus, tasks assigned to the processor with the minimum value of tp* are scheduled first. Note that the value of tp* represents the total time-taken on processor p* assuming all tasks assigned to it could be executed on it in succession without violating any resource constraints, that is, processor p_i is not forced to idle because of unavailable resources that may be needed by the task being scheduled on it. The tasks on a processor are scheduled in the increasing order of their total resource requirements. Let [r'^.r* . . r*,] represent the total resource requirement^* all tasks T such that $L^B(T)$ * 1+1 or $L^F(T)$ = 1 and task T* has been scheduled. The remaining tasks are scheduled by'the following algorithm:

<u>Schedule-Tasks</u>

```
1.. As long as TP is not empty, perform the step
    a. Let s » 1 + max {τ(T_x) + tp_{p_x} | L^B(T_x) » 1+1
    b. For each v, 1 ≤ v ≤ L, let r_v * | r_vx
         where x is such that L^B(T_X) a
         1+1 or L^B(T_X) * 1 and T_x has been scheduled.
    c. Let tp_i be the first element in TP. While •
         is not empty perform the step
         1. Let T_{x_j} be the first element in *_i..
         ii. If for each v, 1≤v≤L, r_v' + r_{vx_j} ≤ S(r_v)
             then let T(T_{x_j}) « s, for each v, let r_v' * r_v' +
             r_{vx_j} and remove T_{x_j} from $_i.
             Task T_{xj} is scheduled on processor P_i and X_{ixj} *
             1 for k * s, s+1,..., s+t_{ix_j}.
    d. Remove tp_i from TP.
2. Form a set of all tasks T_y which have not
    been scheduled and have L^F(T_y) * 1.
3, Repeat this step until $ is empty. If for each
    r_v' + r_vy ≤ S(r_v) and if for some
    i, t_iy ≤ idle time of processor p^, then
    schedule T_y on P_i and remove T_y from +. Else
    remove T_y from ϕ.
```

Once all tasks T_x with $L^B(T_x)$ » 1 have been scheduled, tasks T_y with $L^B(T_y)$ ^ 1-1 are assigned to processors and then scheduled. The steps are repeated until all tasks T_2 with $L^B(T_2)$ * 1 have been scheduled. The specific details and the rules for breaking ties while forming the different sets are described in [19]. Each time an assigned task cannot be scheduled because of insufficient resources, an entry is rcade in a Resource Shortage Table indicating the particular task which could not be scheduled together with units of the particular resource/resources which were needed but were not available. The Resource Shortage Table is used to determine additions to improve the system performance.

3.2 <u>A Heuristic to Solve GSPC</u>

Here we will describe a heuristic for solving the cost constrained problem, GSPC. The heuristic consists of two major steps. The first step is to identify the mix of processors to use such that the cost constraint is not violated. The second step is to solve the resulting GSP by the procedure described in Section 3.3.1. The solution of the second step is used to modify the mix of processors.

The solution procedure works roughly as follows. Initially, all tasks are assumed to be independent and the cost constraint is relaxed. The tasks are assigned to processors using the algorithm assign-tasks. This assignment is used to calculate a_i which represents the value of processor P_i with respect to the assignment. We then solve a knapsack problem ^ given below:

*2 The name comes from a physical manifestation of this single constraint problem - trying to maximize the total value of different sized articles that can be packed into a knapsack.

KSAK:

$$\text{Maximize} \sum_{i \in P} a_i Y_i$$

$$\text{subject to} \sum_{i=1}^{M} CP(P_i)\, Y_i \le COST \quad (8)$$

$$Y_i = 0 \text{ or } 1 \quad \text{all } i \in P$$

Many efficient algorithms are possible to solve Eq. 3-8 because ft has only one constraint [14]. We use a variation of branch and bound to solve it.

The set PI is identified from the solution to KSAK. The tasks are then assigned to the selected set of processors using the algorithm assign-tasks and scheduled on them using the algorithm schedule-tasks. The schedule is used to modify a. and the KSAK is resolved. The steps are repeated until no further improvement results. The details of the procedjre are described in [15, 16].

3,3 An Example

Consider a distributed computer system MP{P,R> with P = {P, P2, P3} and R « {R R2}.

The inputs required, for the task graph of Fig. 3-1 are given in Tables 1, 2 and 3. Table 1 is a matrix representation of G.(Y,E) - the task graph of Fig. 1. Note that the task graph is represented as a 2 *by* |E| matrix called the Edge List Matrix (ELM) such that *if* an edge e.eE Ts incident from node n eV to node n.eV then ELM (l.e^ - n_s and ELM ^{s}(2,e_t) » n^

The values of [t_{ij}] and [r_{ij}] are specified in Tables 2 and 3 respectively. 8(Ri J = 3 units and 8(R_2) = 2 units. L^ • 3.

Fig. 2 is a pictorial representation of the output of the heuristic procedure of Section 3.1.

Note that L £ * 5. Table 4 is the output which indicates ways to decrease the overall execution time by suitable additions to the system. Task T4 could not be started in parallel with T5 and T6 because of resource shortage. It needed 3 units of R, and 2 units of R~ and none *were* available. If we let 8(R_1)=6 and 8(R_2)=4 then L A would be equal to L-

4 A BRIEF DESCRIPTION OF SNONUET'S USES AND FEATURES

The solution procedure outlined in Section 3.3 has been translated into a user friendly, interactive, FORTRAN program called SNONUET. It allows the user to modify the input parameters until either satisfactory execution time is obtained, or no further improvement is possible. SNONUET has been tested on a number of randomly generated examples and it produced near optimal schedules in most cases. The purpose of this section is to illustrate some of the uses and features of SNONUET. To do this we will use a simple example, chosen for explanatory purposes rather than realism.

4.1 Preparation of Input Data for SNONUET

The steps in preparing the input data for SNONUET are described below:

1. Select a level of decomposition and identify a set of primitive tasks, TP, in terms of which to describe the algorithm(s) in question. The primitives can be at various levels. A reasonable way to proceed is to use high level primitives (i.e., relatively large tasks) for the initial design and then refine the design with lower level primitives.
2. Choose the processor and communication network alternatives to be considered.
3. Estimate the time and resource requirement of each primitive task.
4. Estimate the costs of the processors.
5. Prepare the Edge List Matrix, the Execution Time Matrix and the Resource Requirement Matrix as illustrated in Section 3.3.

*-2 An Illustration of the Design Process Using SNOiTUET

We consider the execution of the FuTl Load Flow (FLF) [23] on the distributed computer of Fig. 3. The task graph G.fV.E) of FLF is given in Figure 3 in terms of the primitive task set shown in Table 5.

Three different types of processors are considered, an array processor, AP, (such as the AP 120B) and two special purpose VLSI peripheral processors SP1 and SP2. SP1 does vector sorting operations *wery* quickly. SP2's function is to do L-U factorizations and bdck substitutions quickly.

The estimates of the execution time of the host and the three types of special processors considered are listed in Table 6. Estimates of the per unit costs of the three types of processors are shown in Table 7.

We start with a unibus, distributed computer system shown in Fig. 4 (the communication network is the data channel of the host computer). The motivation for using the common data bus is the simplicity of the interconnection. Also if the communication over the bus does not limit performance, there would be no need to consider more sophisticated interconnection schemes. The unibus of the system is considered to be a resource of the system. The resources corresponding to the communication network are handled in a special way. If the total time needed for all data transfers over the bus of all tasks in G/\(V,E) at any level is found to be more than the user specified percentage of LJ, then the bus is considered to be congested. The details of the bus modelling procedure are described in [18]. SNONUET finds the latest finishing time of all tasks and identifies resource shortages and processor additions to the system which would improve performance. If the unibus of the system is not congested, the number of special processors of a given type may be increased to check if further reduction in the overall execution time is possible. On the other hand, if the unibus turns out to be congested, we introduce another bus amongst the processors sharing the 'congested bus, to relieve the congestton and improve speedup. The above steps are repeated until no further reduction in execution time results.

The output of SNONUET, where there was no cost constraint, indicated that overall execution time could be reduced by increasing the number of APs to 5. The overall execution time obtained by SNONUET has been plotted vs the number of APs in pig. 5 after scaling it so that the stand alone host could sequentially execute FLF in 100 units of time.

The results when a cost constraint is included are shown in Figure 6. As the constraint is tightened, SN'ONUET produces the points along the dotted line (the Pareto Frontier*3). The other points correspond to poorer designs.

5 CONCLUSIONS

This paper has described a systematic procedure that is useful in the selection and design of dedicated distributed computers. The procedure has been coded into an interactive FORTRAN program called SNONUET.

Before SNONUET can be used one must break the algorithm(s) into ordered tasks, select a set of processors for consideration and select a set of resources for the processors to use. One must also estimate the tine and resource requirements for each primitive task. SNONUET will then schedule the tasks on the processors and identify some changes that may be made if further decreases in the overall execution time are to be obtained. One may include the cost of the processors and an upper limit on what the system is to cost. SNON'UET will select a subset of processors and schedule them so as to minimize the execution time of the algorithm(s) and satisfy the cost constraint. This procedure enables us to plot the optimal speedup vs cost for a fixed conr.unication network and would be wery useful when selecting a set of processors from those conroercially available.

The example in Section 5 was chosen more to illustrate the use of SNOfWET than as a realistic design exercise. It could, however, be used for designing distributed computers if they were to be dedicated to solving load flow problems.

Some further work needs to be done to obtain ways to model complicated coircr.unication networks and the queuing delays that result from packet switching. In the present model, for each task requiring the cornmunication network, expected queuing delays are added to the message transmit times. The sum is treated as a deterministic time for which the resource corresponding to the communication network may not be used by another task. This is a major drawback and its remedy may allow us to extend the range of applicability of the procedure to include some asynchronous algorithms.

*3The surface on which the best tradeoffs are obtained between conflicting attributes like speed and cost.

7 REFERENCES

[1] E. 6. Coffman, editor, Computer and Job-Shop Scheduling Theory, Wiley, 1976.

[2] S. Lam and R. Sethi, "Worst Case Analysis of Two Scheduling Algorithms," SIAM Journal of Computing, Vol. 6, 1977, pp. 518-536.

[3] D. K. Goyal, Scheduling Equal Execution Time Tasks Under Unit Resource Restriction, Ph.D7 dissertation, Washington State University, 1976, Computer Science Department.

[4] J. Y. T. Leung, "Bounds on List Scheduling of UET Tasks With Restricted Resource Constaints," Information Processing Letters, Vol. 9, 1979, pp. 167-170.

[5] J. Jaffe, Parallel Computation: Synchronization, Scheduling and Schen-tes, Ph.D. dissertation, Massachusetts Institute of Technology, 1979, Department of Electrical Engineering and Computer Science.

[6] J. D. Ullnan, "NP-Complete Scheduling Problems," Computer and Systems, 1975, pp. 384-393.

[7] J. D. uYli-an, "Polynomial Computer Scheduling Problems," Operating Systems Review, 1973, pp. 96-101.

[8] E. J. Coffman, editor, Complexity of Sequencing Problems, Wiley, Computer and Job Shop Scheduling, 1976, pages 130-164.

[9] J. Bruno, E. J. Coffman, and R. Sethi, "Scheduling Independent Tasks to Reduce Mean Finishing Time," CACM, 1974, pp. 382-387.

[10] M. Held and R. Karp, "A Dynamic Programming Approach to Sequencing Problems," SIAM J. Appl. rath 10, 1, 1962, pp. 196-210.

E". Horowitz and S. K. Sahni, "Exact and Approximate Algorithms for Scheduling Nonidentical Processors," JACH 23, 2, 1976, pp. 317-327.

[12] E. 6. Coffnan, editor, Enurerative and Iterative Computational Approaches, John Wiley & Sons, Inc., Computer Job/Shop Scheduling Theory 1976.

[13] M. J. Krone, Heuristic Prograrming Applied to Scheduling Problems, Ph.D. dissertation, Princeton University, 1970.

[14] John Wiley & Sons, editors, Integer Programming A Wiley-Interscience Publication, 1972.

[15] R. Mehrotra, "Maximum Weighted Patchings on Bipartite Graphs," Technical Report, Carnegie Mellon University, December 1982.

[16] R. Mehrotra, "An Algorithm Based on Finding Weighted Hatchings on Graphs to Schedule Tasks on Multiprocessors," Technical Report, Carnegie Mellon University, December 1982.

[17] J. Edmonds, "Hatching: A Well Solved Class of Integer Linear Programs," Proc. of the Calgary Int. Conf. on Comb. Structures and Their Appl., Gordon and Breach, 1970, pp. 89-92.

[18] J. Edmonds, "Paths, Trees and Flowers," Can. J. Math., Vol. 17, 1965, pp. 449-467.

[19] J. Edmonds, "Maximum Matching and a Polyhedron with 0,1 - Vertices," J. Res. Nat. Bur. Stds., Vol. 69 B, 1965, pp. 125-130.

[20] J. Edmonds, "An Introduction to Matching," Lecture Notes, University of Michigan Summer Eng. Conf.

Table 1. Edge List Matrix for $G_A(V,E)$ of Fig. 3-1.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 4 & 5 & 5 & 6 \\ 4 & 5 & 6 & 7 & 8 & 8 & 9 & 9 \end{bmatrix}$$

Table 3. Resource Requirement Matrix $[r_{ij}]$.

R	A	T1	T2	T3	T4	T5	T6	T7	T8	T9
R1	3	1	1	1	3	1	0	2	0	3
R2	2	0	1	1	2	2	0	1	0	0

A = Amount of Resource.

Table 2. Executive Time Matrix $[t_{ij}]$.

	T1	T2	T3	T4	T5	T6	T7	T8	T9
P1	2	2	1	1	1	1	3	1	1
P2	3	1	1	3	1	2	1	2	1
P3	1	1	2	4	3	1	4	1	3

Table 4. Resource Shortage Table.

Task	R1	R2
T4	3	2

Table 5. A Set of Primitive Tasks to Describe FLF of Figure 3.

Task	Symbol
No operation (used for synchronization)	T_ϕ
Vector Add ($c_i=a_i+b_i$, $i=1,n$)	VA
Vector Inner Product ($c=\sum_{i=1}^{n} a_ib_i$)	VP
Vector Scale ($b_i=c*a_i$, $i=1,n$)	VS
Vector Divide ($b_i=a_i/c$, $i=1,n$)	VD
Vector Multiply ($c_i=a_ib_i$, $i=1,n$)	VM
Vector Sort (arrange elements in increasing order)	V Sort
LU Factorization	LUF
Back Substitution	BS

Table 7. Cost Estimates.

Processor	Cost
AP	50
SP1	10
SP2	20
HOST	0

Table 6. Estimate of Execution Time.

	Estimates of Task Execution Times			
Primitive Task	HOST	AP	SP1	SP2
T_ϕ	0	0	0	0
VA	100	35	∞	∞
VP	1100	410	∞	∞
VS	1000	340	∞	∞
VD	1000	340	∞	∞
VM	1000	340	∞	∞
V Sort	4950	5000	700	∞
LUF & BS	4000	3000	∞	400

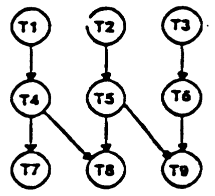

Figure 1. A Task Graph $G_A(V,E)$.

P1	T3		T4	T9	
P2	T2	T5			T7
P3	T1	T6		T8	
Time Unit	1	2	3	4	5

Figure 2. Pictorial Representation of the Schedule.

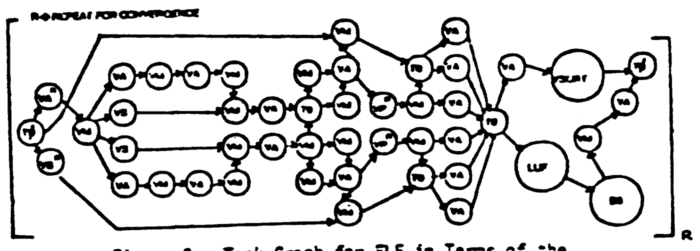

Figure 3. Task Graph for FLF in Terms of the Primitive Task Set of Table 5.

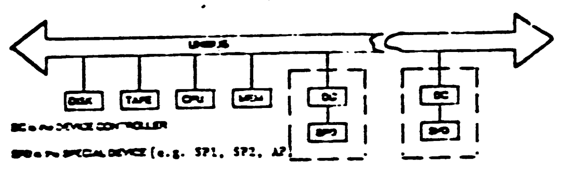

Figure 4. A Unibus Distributed Computer System.

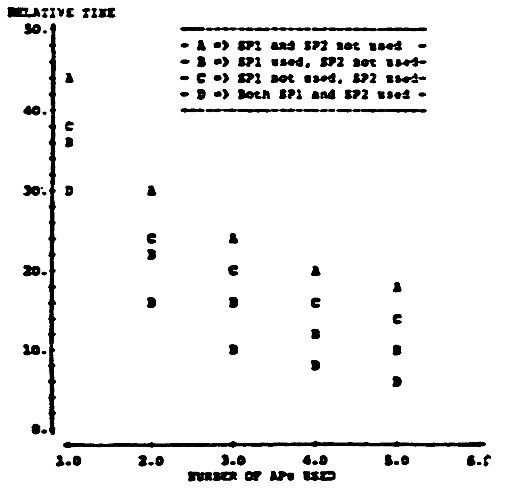

Figure 5. Execution Times Obtained by SNCNUET.

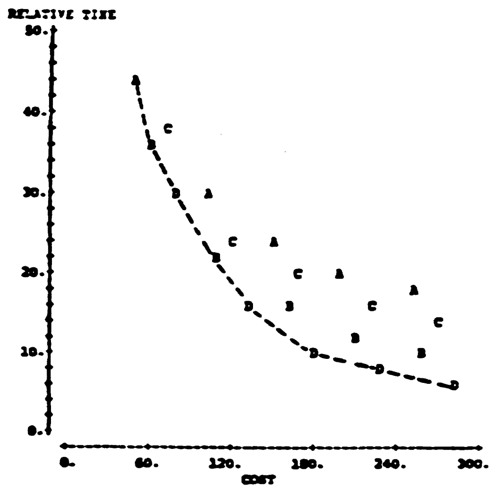

Figure 6. Plot of Relative Time vs Cost. Points on the broken line were obtained by SNCNUET.