WORKLOAD SCHEDULING FOR TASK GRAPHS WITH COMMUNICATION COSTS ON PARALLEL SYSTEMS

Technical Report CSE-91-11

Carolyn McCreary

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

Yahui Zhu

Department of Computer Science and Engineering
North Dakota State University
Fargo, ND 58105-5075

September 1991

Workload Scheduling for Task Graphs with Communication Costs on Parallel Systems

Yahui Zhu
Department of Computer Science, North Dakota State University, Fargo, ND 58105-5075
and
Carolyn McCreary
Department of Computer Science and Engineering, Auburn University, AL 36849

Abstract

In this paper we extend the classical work on multiprocessor scheduling in several major ways. We define a new notion of scheduling, called the *workload schedule* which captures the essence of the communication/computation trade-off. It is a mechanism by which optimal schedules can be generated while considering both execution and communication costs.

We extend the theory of scheduling by proving that the parallel scheduling of programs with tree dependence structures is an NP-complete problem when arbitrary communication and execution costs are assigned and when the processes are scheduled on an unbounded number of processors. This contrasts sharply to the situation that occurs if communication costs must be bounded by execution costs. In this case, an optimal schedule can be found in O(n) time [PY90].

We also present three algorithms for scheduling trees efficiently. The algorithms range from determining the optimal schedule, intractable in the general setting, to an efficient greedy algorithm with a sound heuristic basis.

1. Background

A sequential program can be represented as a directed acyclic graph (DAG). Each vertex in a DAG denotes a task with a processing time; each edge denotes the precedence relation between the two tasks and the weight of the edge is the communication cost which incurs if the two tasks are assigned to different processors. Given a program DAG, we can partition the DAG into grains of appropriate size and assign the grains to processors of a parallel machine to shorten the execution time of the program. The partitioning and assignment is called the *scheduling* problem. The problem is also known as *grain size determination* in [KL88], *clustering problem* in [KB88, Yu84], and *internalization pre-pass* in [Sa89]. The problem is important because solution methods can be used to generate efficient parallel programs.

In classical DAG scheduling, communication costs are not considered [Co76, GLLK79]. Introducing the communication cost is necessary because communication between processors does take time in real parallel systems, especially in distributed memory systems where communication costs tend to be high relative to processor speed. Researchers and practitioners in parallel computation often emphasize two conflicting goals: obtaining a high degree of parallelism and minimizing inter-processor communication. As the degree of parallelism increases, communication costs also increase, and communication costs can be minimized only by removing

all concurrency. The challenge in the extended scheduling problem is to consider the trade-off between communication and communication [PU87].

Many researchers have studied the problem and proposed solutions. Based on the techniques employed, the earlier methods can be classified into the following three categories:

- Critical path heuristics [GVY90, GY90, KB88, Sa89, WG90, Yu84]: Extending the critical path method due to Hu [Hu61] in classical scheduling, these algorithms try to shorten the longest execution path in the DAG. A comparison study of these methods can be found in [GVY90].
- List scheduling heuristics [AHC90,HCAL89,KL88,LHCA88, Li90,PY90, RL90]: These algorithms assign priorities to the processes and schedule them according to a list priority scheme. Extending the list scheduling heuristic due to [Gr69] in classical scheduling, these algorithms use greedy heuristics and schedule tasks in a certain order. Task duplications have been used in [KL88,AHC90, RL90] to reduce the communication costs.
- Graph decomposition method [MG89, MG90]: Based on graph decomposition theory, the method parses a graph into a hierarchy (tree) of subgraphs. Communication and execution costs are applied to the tree to determine the grain size that results in the most efficient schedule.

Among these three approaches, the graph decomposition method seems to be the most robust and flexible. Rather than modifying the structure of a DAG or scheduling tasks from a priority list as used in the other two approaches, it derives the parallelism by analyzing the structure. The method is based on graph theory, and is both general and conceptually simple.

In this study a new concept for systematic exploitation of parallelism in DAGs is introduced. A *workload* records the busy time intervals needed by the tasks executed on a processor. Each DAG vertex will be associated with a processor and the processor's current list of scheduled busy times. The workload provides a way to capture the communication time trade off during the scheduling process. Based on the workload concept, a *workload scheduling* technique is presented whose goal is to minimize total elapsed time.

In this paper the technique is applied to schedule tree DAGs. The background definitions and assumptions are given in section 2. The concept of the workload is defined in section 3. Section 4 describes the underlying scheduling algorithms and presents the optimal scheduling algorithm. Section 5 surveys the NP-Completeness property for multiprocessor scheduling and presents a new NP-completeness result. Section 6 describes more efficient scheduling algorithms and Section 7 summarizes our work.

2. Preliminaries

Let a DAG be represented by a weighted graph G = (V,E), where V is the set of vertices and E is the set of edges. Each vertex $v \in V$ represents a task with a task time t(v). Each directed edge $(u,v) \in E$ represents a precedence relation, whose weight c(u,v) denotes the communication cost. For an edge (u,v), u(v) is called the *predecessor (successor)* of v(u). If a vertex has no predecessor (successor), then it is called a *source (sink)* vertex. If there is a path from vertex u to v then u(v) is called the *ancestor (descendant)* of v(u). The *level* of a vertex is the number of edges in the longest

path from a source vertex to that vertex. (source vertices are at level 0). The degree of a vertex is the number of edges incident to the vertex. The *indegree* (*outdegree)* of vertex u is the number of edges (v,u), ((u,v)). An *intree* (simply called a tree in this paper), is a connected DAG with n vertices and n-1 edges where the out degree of any vertex is no greater than 1.

A *schedule* of a DAG is a mapping of the tasks onto processors such that all predecessor tasks have executed, the necessary communication has taken place, and no interruption is allowed during task execution (i.e. tasks are nonpreemptive). Unlike schedulers that may activate multiple executions of the same task, we will create schedules where no task is allowed to run on different processors. The *finish time* of a schedule is the time when all the tasks are completed. A schedule with the minimum finish time is called an *optimal schedule*.

We will assume a virtual parallel machine model that satisfies the following conditions:

- The number of processors is unbounded and the processors are fully connected.
- Communication protocols are error free; any number of processors may communicate with any others simultaneously; and the cost of communication between any 2 processors is the same as between any other 2 processors.
- Communication and execution may occur simultaneously.
- All messages are sent at the time of completion of the originating task, and the receiving task cannot begin execution until all messages are received from all preceding tasks.

We make this model more explicit by associating execution and communication times for each vertex and edge. A *tc-pair* ((t(u),c(u)) will represent vertex *u*'s task execution time and communication cost to its successor. When vertices are aggregated to execute on the same processor, the model assumes the execution times are additive. When vertex *w*'s predecessors *u* and *v* execute in parallel, w will be assigned to the same processor as either *u* or *v*. Assume that *w* is assigned to the same processor as *u*. *w*'s execution cannot begin until *u* is finished and *v* has passed its results to *u*'s processor.

The advantage of this model is that is allows researchers to concentrate on the principles needed to exploit the parallelism rather than on the details of the machine architectures. Many researchers have recognized the importance of the model[PY90]. Once the parallelism of an application is exploited, a mapping method can be employed to map the parallelism onto real parallel architectures. This approach has been taken by several recent research projects, such as the SISAL parallel compiler [Sa89], the Hypertool parallel programming environment [WG90], and the parallel code generation system in [TDP89].

3. The Workload Concept

In a binary tree, two sibling tasks can be scheduled in the following ways: (i) parallelize them by executing them on two separate processors, or (ii) aggregate them by executing them both on one processor. Choice (i) leads to a communication delay while the results of the two processes are gathered by one of the processors, while choice (ii) implies that both processes complete their execution before additional processing can continue. In workload scheduling a *workload schedule* is defined at each DAG vertex. This schedule contains two types of information: a sequence of

busy intervals and a *start time*. The sequence of *busy intervals* represents the times when the processor containing the DAG vertex is executing some code, while the start time gives the time when the DAG vertex can begin its execution. If a communication delay is required, it will be accounted for in the start time.

Definition: The *workload* of vertex u, denoted by W(u), is defined as a list of k time intervals $[a_i,b_i]$ and a starting time s(u), as follows:

$$W(u) = ([a_1,b_1],[a_2,b_2], ...[a_k,b_k], s(u)), \quad \text{where}$$
$$a_1 = 0,\ a_i <= b_i \text{ and } b_i < a_{i+1} \text{ for all i and } b_k <= s(u).$$

The processor executing u is busy from a_i to b_i. An empty workload is denoted by ([0,0],0).

For example, the workload: ([0,5],[10,25],35) shows that a processor is busy from 0 to 5 and 10 to 25, idle from 5 to 10 and 25 to 35, and ready to execute at time 35. A set of workloads may be associated with each vertex to represent several possible choices for parallelization and aggregation of the ancestor vertices. When there is more than one workload for vertex u, a second identifier, j, can be attached to the workload identifier as in W(u,j).

The concept of workload scheduling is to keep track of the workloads on the *critical path* to the root, the critical path, and to make decisions at each vertex of the path that will minimize the total elapsed time. Knowledge of the busy intervals implies knowledge of the idle time and attempts are then made to fill these gaps with real work.

Consider the tree of Figure 1 whose vertices (edges) are labeled with their execution (communication) times. If *a* and *b* are executed in parallel, the corresponding processors are busy for time intervals [0,20] and [0,14], respectively. If *u* is placed on the same processor as *a*, the execution of *u* cannot begin until time 34, when *b* has completed and communicated its results to *u*'s processor. If *u* is placed on the same processor as *b*, its execution cannot begin until time 35. If *a* and *b* are executed on the same processor, the model defines the execution times as additive, the busy intervals are [0, 14][14,34] and vertex u can begin execution immediately at time 34.

If above example is extended to a tree of depth 3, as shown in Figure 2, consider the possible schedules for vertex *w*. Let A indicate aggregate and P*x* indicate parallelize and place *x* on the same processor as *w*. A string $S_1S_2S_3$ identifying workload $W(z,S_1S_2S_3)$ means the choice S_1 is made at vertex *z* when the workload of the left ancestor is S_2 and the workload of the right ancestor is S_3. For each pair of workloads from ancestors *u* and *v*, there are three choices for S_1: A, P*u* and P*v*. Since there are 9 *u,v* pairs, there are 27 possible workloads at vertex *w*. To illustrate their construction, consider W(*w*,P*u*P*a*P*c*). P*u*P*a*P*c* indicates workloads W(*u*,P*a*) and W(*v*,P*c*) are to be executed in parallel and *w* is to be placed on the same processor as *u*. The resulting workload includes W(*u*,P*a*), the busy time representing the work of vertex *u*, and a start time that follows the execution of *v* and the communication of its results (s(*v*)+t(*v*)+c(*v*)):

W(*w*,P*u*P*a*P*c*)=([0,20][34,39],89)

Aggregation combines all workloads, utilizing idle times whenever possible.

W(*w*,AP*b*P*c*)=([0,14],[14,34],[34.39],[39,44],44)

The optimal schedule is shown in Figure 3.

The above example shows how the communication-time trade-off has been captures in a workload: the time intervals are the task execution times, while the gaps between the intervals are

communication delays. As the scheduling proceeds, workload aggregations allow these gaps to be used productively. In general, the larger the communication costs, the more effective the use of these gaps through aggregation and workload scheduling. Heuristics will be developed to prune the choices to a reasonable set and select schedules that will result in early completion times.

4. Workload Scheduling

In this section, algorithms will be presented to develop an optimal schedule based on the workload concept. A dynamic programming algorithm can be used to find the optimal workload of the root, yielding an optimal schedule for the entire tree. The general procedure is to start from the source vertices with empty workloads and compute all workloads that can produce an optimal schedule for the other vertices, level by level. In a binary tree, the workloads for an interior vertex are calculated by considering the two available options: parallelize or aggregate the predecessors. Once the workloads of the root have been computed, the final schedule of the DAG is derived from the workload with the earliest start time.

4.1 Algorithms for Aggregation and Parallelization

Three algorithms are given for binary trees: UPDATE which incorporates the execution requirements of the current vertex, AGGREGATE which generates the workload created by aggregating the predecessor vertices, and PARALLELIZE which generates the workloads created by parallelizing the predecessor vertices. Algorithm UPDATE takes as input a workload, W(u) and a task time for *u*, t(u). It produces a workload with task time t(u) added after s(u). If $s(u) = b_k$, then the last interval is expanded to include u's task time. Otherwise a new interval is created. In either case, s(u) is updated.

Algorithm **UPDATE**(W(u), t(u));
/* Input: a workload $W(u) = ([a_1,b_1],[a_2,b_2], ...[a_k,b_k], s(u))$ and a task time t(u). */
/* Return: the modified workload W with a task time t(u) added after time s(u) */

If $b_k = s(u)$ then $b_k = b_k + t(u)$; $s(u) := b_k$ /* extend last interval */
else $k := k + 1$; $a_k = s(u)$; $b_k = s(u) + t(u)$; $s(u) := b_k$ /* add new interval */
Return W(u)

Parallelize

When *u*'s ancestors are executed in parallel, the workload computed for *u* is a copy of an ancestor's updated workload with a start time that reflects the delay waiting for the other ancestor to transmit its results. For example in Figure 2., W(w,PuPaPc) is updated W(u,Pa) with start time 34+t(v)+c(v). When parallelizing the predecessors of vertex *u*, it is tempting to assign *u* to the processor on which it can begin execution earliest. In Figure 2, for example, a local decision at *u* to minimize the start time would place *u* on the same processor as *a* and assign *u* the workload ([0,20],34). If *u* then could be aggregated with W(*v*,P*c*) the workload would be ([0,50],50). On the other hand, placing *u* with *b* creates workload ([0,14],35) and merging with W(*v*,P*c*) gives

workload W(*w*,AP*b*P*c*) = ([0,44],44). The length and timing of the busy intervals, as well as the start times, play a role in creating the optimal schedule. The PARALLELIZE algorithm produces both possible workloads for a vertex with two predecessors.

```
Algorithm PARALLELIZE (W(x), t(x), c(x), W(y), t(y), c(y));
/* Input: workloads W(x) and W(y), task times t(x) and t(y), communication
requirements, c(x) and c(y). */
/* Output: workloads W(u,Px) and W(u,Py) for vertex u with predecessors x and y
*/
 /* update start times */

UPDATE( W(x),t(x)); UPDATE (W(y),t(y));
 /* x and u on same processor - use x's workload */
    W(u,Px) = W(x)
    s(u,Px) = max(s(x),s(y) + c(y))

 /* y and u on same processor - use y's workload */
    W(u,y) = W(y)
    s(u,Py) = max(s(x) + c(x),s(y))
Return W(u,Px), W(u,Py)
```

Aggregate

Aggregating two workloads, W(x) and W(y), produces a third workload, W(u,A) that includes all the busy times from W(x) and W(y). Since the underlying architectural model assumes processing is additive, the busy intervals are created by combining busy times. When aggregating two workloads with parallel predecessors, only the work referred to by the busy times are merged. No attempt is made to combine the parallel work. For example, suppose in Figure 2, W(u,Pa) and W(v,Pc) are aggregated to form W(w,APaPc). Vertices *a,u,c,v* and *w* are executed on the critical processor, say #1. Meanwhile *b* and *d* are assigned to two different processors, say #2 and #3. No attempt is made to merge *b* and *d*.

If two busy times overlap, the total execution time is included in an interval from the start of the first, t_f, to t_f plus the sum of the execution times. i. e. if $[a_i, b_i]$ is to be merged with $[c_j, d_j]$ and $a_i \leq c_j \leq b_i$, the resultant interval is $[a_i , a_i + (b_i - a_i + d_j - c_j)]$. As new intervals are formed, overlapping may occur with original intervals resulting in new merged intervals.

The procedure AGGREGATE is based on the standard merge algorithm.

```
Algorithm AGGREGATE ( W(x), t(x), W(y), t(y));
/* Input: workloads W(x) and W(y), task times t(x) and t(y) */
/* Output: the aggregated workload W(u) = ([e1,f1], ..., [er,fr], s(u)) */

UPDATE( W(x),t(x)); /* denote W(x) = ([a1,b1],[a2,b2], ...[ap,bp], s(x)) */
```

UPDATE (W(y),t(y)); /* denote W(y) = ([c_1,d_1],[c_2,d_2], ...[c_q,d_q], s(y))

Merge intervals from both workloads W(x) and W(y) until all intervals from one workload are included.
Insert the remaining busy intervals.
Set start time to end of final interval.

For example: Suppose W(x) = ([0,2],[3,5],10), t(x) = 2, W(y) = ([0,1],[7,8],[11,12],14) and t(y)=2. Then merging [0,2],[0,1] and [3,5] gives [0,5]; [10,13] includes x's execution and [11,12]; [14,16] represents y's execution. The merged workload is W(u) = ([0,5],[7,8],[10,13],[14,16],16).

The algorithm AGGREGATE scans through both workloads once, so it runs in linear time. The resulting start time is minimum when two processes are aggregated. The PARALLELIZE algorithm copies two workloads, again yielding a linear time algorithm.

4.2 Subschedules

The workload at task u, W(u), describes the busy intervals and next available execution time for the processor on which task u will execute. Workloads are created as we traverse the tree and combine the work done by previous processing. If all potential parallelism were exploited, the total number of processors used would equal the number of leaf tasks and with each step down the tree, fewer processors would be required. At the root, only one processor is required. To use the processors most efficiently, it is helpful to place heavier execution burdens on processors that will be released as we go down the tree and to place lighter execution loads on the subtree roots. In this way there are greater opportunities for aggregation that will not lengthen the root's workload. For example for task u with t(u) = 3, consider workloads W(u,1) = ([0,8], [10,15], 17)and W(u,2) = ([0,4], [10,12],17). The updated workloads (including u's execution requirements) are $\overline{W}$(u,1) = ([0,8], [10,15], [17,20], 20) and $\overline{W}$(u,2) = ([0,4], [10,12],[17,20],20). Both have the same start time, but $\overline{W}$(u,2) is more lightly loaded. Suppose an attempt is made to aggregate u's workloads with updated W(v) = ([0,5], [10,16], 16). W(v) aggregated with W(u,1) gives the workload ([0, 27],27), and W(v) aggregated with W(u,2) gives the workload ([0,9],[10,21],21). The idle time represents communication delay which can be utilized when aggregation occurs. This idea is the basis upon which subschedule pruning is based.

Definition: Given two workloads, W(x,1) = ([a_1,b_1],[a_2,b_2], ...[a_p,b_p], $s_1(x)$) and W(x,2) = ([c_1,d_1],[c_2,d_2], ...[c_q,d_q], $s_2(x)$), then workload W(x,1) is said to be a ***subschedule*** of W(x,2) if for all i, $1 \le i \le p$, there exists some j, $1 \le j \le q$ such that $[a_i,b_i] \subseteq [c_j,d_j]$ and $s_1(x) \le s_2(x)$. W(x,1) is a *proper subschedule* of W(x,2) iff $[a_i,b_i] \subseteq [c_j,d_j]$ as before and $s_1(x) < s_2(x)$ or $[a_i,b_i] \subset [c_j,d_j]$ for some i,j pair and $s_1(x) \le s_2(x)$. When W(x,1) is a subschedule of W(x,2), W(x,2) is said to be a *superschedule* of W(x,1).

In the example above W(u,2) is a subschedule of W(u,1).

Lemma 1: Let W(u,1) and W(u,2) be two workloads at vertex *u* and let W(w,1) and W(w,2) denote the workloads obtained by aggregating W(u,1) and W(u,2) resp. with W(v). If W(u,1) is a subschedule of W(u,2) then W(w,1) is a subschedule of W(w,2).

Proof: Follows directly from the procedure AGGREGATE.

Let W(w,ix) denote the workload at w resulting from parallelizing W(q,i) with W(v) and placing vertex x on the processor with w.

Lemma 2: Let W(w,1u) and W(w,1v) be the workloads obtained by parallelizing W(u,1) and W(v). W(w,2u) and W(w,2v) denote the workloads obtained by parallelizing W(u,2) and W(v). If W(u,1) is a subschedule of W(u,2), then W(w,1u) is a subschedule of W(w,2u) and W(w,1v) is a subschedule of W(w,2v).

Proof: Follows directly from the procedure PARALLELIZE.

4.3 An Optimal Solution

At each vertex in the binary tree, exactly three choices are presented: aggregate by placing the predecessors on the same processor or parallelize the predecessors and place the current vertex with the left (right) predecessor. Workloads that are superschedules of the other workloads can be eliminated from consideration. By keeping track of all remaining choices and the corresponding workloads at each vertex, an exact solution to the problem can be found. At each vertex, all pairs W(x,i) and W(y,j) of workloads from the predecessors x and y are combined and both the aggregation and parallelization algorithms are applied.

```
Algorithm OPTIMAL SCHEDULE(T:labeled tree):
/* Input: tree with task times, t(x), and communication times c(x) associated with
each vertex */
/* Output: optimal workload schedule of root, r. */
/* Let L be the level of the root */
For lev = 1 to L do
    For each vertex u on level lev do
        index = 1; temp = 0;
        /*let x and y denote the predecessors of u and n(v) the number of workloads
          associated with vertex v*/
        For i = 0 to n(x) - 1
            For j = 1 to n(y)
                W(u, index)) = AGGREGATE(W(x,i), t(x), W(y,i), t(y))
                If W(u,index) is not a superschedule of W(u,k), 1<=k<index
                    then index + index+1
                W(u,index)), W(u,index+1) =
                    PARALLELIZE(W(x,i),t(x),c(x),W(y,j),t(y),c(y))
                If W(u,index) is not a superschedule of W(u,k), 1<=k<index
                    then temp = 1
                If W(u,index+1) is not a superschedule of W(u,k), 1<=k<index
                    then index = index+1+temp else index = index + temp
        n(u) = index
Among workloads, W(r,i), select the one with smallest starting time and call it W(r)
Return (W(r))
```

The final schedule is completed in time s(r) + t(r), the start time of W(r) plus the task time for the root vertex.

Theorem 3: The OPTIMAL SCHEDULE algorithm gives a schedule with minimal completion time.

Proof: The algorithm finds all possible schedules except those eliminated through the subschedule pruning. But Lemmas 1 and 2 imply that superschedules can never generate an earlier start time. []

For a complete binary tree of depth 3 with $n = 2^{d+1} - 1$ vertices, the number of workloads at the root vertex is $O(3^n)$. Although this dynamic programming solution is intractable, it does give an optimal schedule. In the following sections, it will be shown that finding the optimal schedule is an NP-complete problem, and a series of algorithms will be developed that produce good solutions that may not be optimal. The algorithms use the basic ideas of the dynamic programming approach, but eliminate from consideration some of the workloads at each vertex. Three methods of pruning are considered.

5. NP-Completeness

In Section 4., an intractable algorithm to find an optimal schedule was given. The natural question arises as to whether a polynomial time algorithm exists for tree scheduling in this model. NP-completeness results for scheduling trees vary drastically in very similar settings. When the number of processors is fixed, finding an optimal schedule of trees, even without considering communication costs, is NP-Complete[GJ79]. On the other hand, if the machine model assumes an unbounded number of (virtual) processors and ignores communication costs, then scheduling trees is in P. Furthermore Anger, Hwang and Chow [AHC90] show that if communication delays are assumed to be no longer than the shortest task processing time, there is a linear-time optimal algorithm for tree scheduling. The variable complexity of the problem emphasized the need to pay careful attention to the complexity of scheduling under new machine models.

Papadimitriou and Yannakakis [PY90] prove that when the execution costs of the vertices are constant and the communication cost between two vertices is constant, scheduling a DAG with possible repetitions of the vertices on an unlimited number of processors is an NP-complete problem. We extend these results by proving that even for trees with arbitrary communication and execution costs, when there are an unbounded number of processors, scheduling is an NP-complete problem. This result is in sharp contrast with that in [AHC90]. By relaxing the coarse-grain condition, the difficulty of the tree scheduling problem changes from linear-time solvable to NP-complete.

To prove NP-completeness, we reduce the partition problem, which is a well-known NP-complete problem [GJ79,] to our tree scheduling problem. The partitioning problem states that, given a set of numbers, it is an NP-complete problem to find a subset whose values sum to half the sum of all the values in the set.

Theorem 4: It is an NP-complete problem to decide, given a time T* and a tree whose vertices and edges are assigned arbitrary execution and communication times, whether there exists a schedule on a machine with an arbitrary number of processors such that its finish time is no greater than T*.

Proof: Given an instance of the partition problem, $B = \{v_i: 1 \le i \le n, v_i \text{ a positive integer}\}$, let $A = (\Sigma v_i)/2$. The ordered pair, (t,c), (called a tc-pair), will represent a vertex's task time and communication cost to its successor. We will construct a tree dag that corresponds to this instance of the partition problem, and we will show that finding the optimal schedule will correspond to finding the subset of B whose values sum to A. The construction is illustrated in Figure 4. The root has n subtrees which correspond to the *n* values in B. The *i*th subtree is rooted at w_i, with tc-pair $(1/(2n^2),A^3)$. Vertex w_i has three predecessors, denoted as p_i, q_i and r_i, with corresponding tc -pairs $(v_i,A-v_i)$, $(A^2+A-1,1)$ and $(1/(2n^2),A-1/(2n^2))$, respectively. Vertex p_i has two predecessors, a_i and b_i with tc-pairs $(1/(2n^2),A^2-1/(2n^2))$ and $(A^2-1/(2n^2),1/(2n^2))$. Vertex r_i has two predecessors, c_i and d_i with their tc-pairs (Av_i,A^2-Av_i) and $(A^2-1,1)$. Clearly the tree can be built in polynomial time.

The possible workloads at p_i include

$W(p_i,1) = ([0,A^2],A^2)$ (aggregation)

$W(p_i,2) = ([0,1/(2n^2)],A^2)$ (parallelize and place p_i on same processor as a_i)

$W(p_i,3)= ([0,A^2-1/(2n^2)],A^2)$ (parallelize; place p_i on same processor as b_i)

The possible workloads at r_i include

$W(r_i,1) = ([0,A^2+Av_i-1],A^2+Av_i-1)$ (aggregation)

$W(r_i,2) = ([0,Av_i],A^2)$ (parallelize and place r_i on same processor as c_i)

$W(r_i,3) = ([0,A^2-1],A^2)$ (parallelize and place r_i on same processor as d_i)

$W(p_i,2)$ is a subsequence of p_i's other workloads and $W(r_i,2)$ is a subsequence of r_i's other workloads. By lemmas 1 and 2, the only workloads that will produce optimal schedules at w_i are $W(p_i,2)$ and $W(r_i,2)$.

The possible optimal workloads at w_i include:

(parallelize all 3 predecessors)

$W(w_i,1) = ([0,1/(2n^2)],[A^2,A^2+v_i],A^2+A)$ (place w_i on same processor as p_i)

$W(w_i,2) = ([0,Av_i],[A^2,A^2+1/(2n^2)],A^2+A)$ (place w_i on same processor as r_i)

$W(w_i,3) = ([0,A^2+A-1],A^2+A)$ (place w_i on same processor as q_i)

When considering aggregation, notice that the load at q_i is so large that when aggregated with either p_i or q_i, the start time and processing requirements will be greater than $A^2 + A$ and it will be a superschedule of the others. When p_i and r_i are aggregated and parallelized with q_i, the resulting workload is:.

$W(w_i,4) =([0,Av_i+1/(2n^2)],[A^2,A^2+v_i+1/(2n^2)],A^2+A).$

By lemmas 1 and 2, the optimal schedule is produced by pruning all workloads that are superschedules to some schedule. The pruning leaves two choices of workloads at w_i:

$W(w_i,1) = ([0,1/(2n^2)],[A^2,A^2+v_i],A^2+A)$ and

$W(w_i,2) = ([0,Av_i],[A^2,A^2+1/(2n^2)],A^2+A)$.

The communication costs to the root are too large to benefit from executing any two vertices concurrently. Aggregating all vertices w_i produces the optimal schedule.

Let S be the subset of vertices $w_{i_1} \ldots w_{i_k}$ for which the workload $W(w_i,2)$ is chosen. Let |S| denote $\sum_{j=1}^{k} v_{i_j}$ corresponding to w_{i_j} in S. The total processing requirements for the workload at the root is given by:

$$T = A*|S| + k(1/(2n^2)) + (n-k)(1/2n^2) + 2A - |S| + n/(2n^2) \qquad *$$

The first two terms of T come from the busy intervals in $W(w_i,2)$; the third and fourth term from the busy intervals in $W(w_i,1)$ and the last term corresponds to the processing required at the w_i vertices. If the start time of the root is T, then the choice of S gives an optimal schedule. We show that only if |S| = A, the optimal schedule is achieved. If |S|=A, $T= A^2 + A + 1/n$. Since B is a set of integers, if |S|>A, then the first term in equation * will be greater than $A^2 + A$ and $T > A^2 + A + 1/n$. If |S| < A then 2A- |S| >A, and since the second busy interval of both $W(w_i,1)$ and $W(w_i,2)$ cannot start until time A^2, $T > A^2 + A + 1/n$. On the other hand, if |S| = A, T is achieved by scheduling all busy times from $W(w_i,2)$ followed by all busy times from $W(w_i,1)$ and then all w_i processing.

We have reduced a case of the optimal scheduling problem in this setting to the partition problem, and thereby proved that scheduling trees with arbitrary communication costs is NP-complete. [].

6. Pruning Techniques

The optimal dynamic programming solution to the scheduling problem of Section 4. is intractable for a general task graph. For graphs will no possibilities of finding subschedules, algorithm will compute $O(3^n)$ workloads. In section 5 we proved that there is no reasonable hope of finding a polynomial scheduling algorithm for trees with communication overhead. In this section we offer several algorithms that will execute in a reasonable time and produce good results.

6.1 Lighter Loads

The subschedule relationship was defined to reduce the load on the critical path processor so that processing time was available when needed for aggregation. This idea is extended to comparing workloads where one's scheduled busy times are not necessarily subintervals of the other's.

> **Definition:** Given two workloads, $W(x,1) = ([a_1,b_1],[a_2,b_2], \ldots [a_p,b_p], s_1(x))$ and $W(x,2) = ([c_1,d_1],[c_2,d_2], \ldots [c_q,d_q], s_2(x))$, W(x,1) is said to be ***lighter*** than W(x,2) iff $\sum(b_i-a_i) \leq \sum(d_i-e_i)$ and $s_1(x) \leq s_2(x)$, i.e. the total busy time from W(x,1) ≤ the total busy time from W(x,2) and the start time of W(x,1) is less than the start time of W(x,2). W(x,1) is a ***strictly lighter***

load if $\sum(b_i-a_i) < \sum(d_i-e_i)$ and $s_1(x) \leq s_2(x)$, or $\sum(b_i-a_i) \leq \sum(d_i-e_i)$ and $s_1(x) < s_2(x)$. W(x,2) is said to be ***heavier*** than W(x,1).

For example, let W(v,1) = ([0,4],[8,10],10) and W(v,2) = ([0,3],[4,7],[9,10],10). W(v,1) is lighter than W(v,2) since the total busy times of W(v,1) and W(v,2) are 6 and 7 respectively. Notice that W(v,1) is not a subschedule of W(v,2).The subschedule relationship is stronger than the lighter relationship. Whenever W(u,1) is a subschedule of W(u,2), it is also true that W(u,1) is lighter than W(u,2).

A reasonable algorithm heuristic algorithm applies this pruning to the OPTIMAL SCHEDULE algorithm:

```
Algorithm LIGHTER_LOAD_SCHEDULE(T:labeled tree):
/* Input: tree with task times, t(x), and communication times c(x) associated with
each vertex */
/* Output: workload schedule of root, r. */
/* Let L be the level of the root */
For lev = 1 to L do
    For each vertex u on level lev do
        index = 1; temp = 0;
        /*let x and y denote the predecessors of u and n(v) the number of workloads
          associated with vertex v*/
        For i = 0 to n(x) - 1
            For j = 1 to n(y)
                W(u, index)) = AGGREGATE(W(x,i), t(x), W(y,i), t(y))
                If W(u,index) is not a lighter load than W(u,k), 1<=k<index
                    then index + index+1
                W(u,index)), W(u,index+1) =
                    PARALLELIZE(W(x,i),t(x),c(x),W(y,j),t(y),c(y))
                If W(u,index) is not a lighter load than W(u,k), 1<=k<index
                    then temp = 1
                If W(u,index+1) is not a lighter load than W(u,k), 1<=k<index
                    then index = index+1+temp else index = index + temp
        n(u) = index
Among workloads, W(r,i), select the one with smallest starting time and call it W(r)
Return (W(r))
```

6.2 Parallel Decision Pruning

One of the most intuitive pruning choices comes when two vertices are to be parallelized. Instead of considering two workloads resulting from parallelizing u and v, choose the one that results in the earliest start time. The parallelize algorithm with pruning then becomes:

```
Algorithm PPARALLELIZE (W(x), t(x), c(x), W(y), t(y), c(y));
/* Input: workloads W(x) and W(y), task times t(x) and t(y), communication
requirements, c(x) and c(y). */
```

```
/* Output:  a single workload W(u) for vertex u with predecessors a and b */

UPDATE( W(x),t(x)); UPDATE (W(y),t(y));
            /* updated start times include task times */
If s(x) + c(x) > s(y) + c(y) then
          /* a and u on same processor - use a's workload */
            W(u) = W(x)
            s(u) = s(y) + c(y)
else
      /* b and u on same processor - use b's workload */
            W(u) = W(y)
            s(u) = s(x) + c(x)
Return W(u)
```

The PPARALLELIZE algorithm can be used place of the PARALLELIZE algorithm in the LIGHTER_LOAD algorithm to produce an even faster algorithm, though its result may be less accurate.

6.3 Local Greedy Decision

All workload choices except one can be eliminated to produce an O(n) algorithm when scheduling overhead must be minimized. Such a greedy algorithm that makes local decisions will choose the schedule that has the smallest start time.

```
Algorithm GREEDY_SCHEDULE(T:labeled tree):
/* Input: tree with task times, t(x), and communication times c(x) associated with
each vertex */
/* Output: workload schedule of root, r. */
/* Let L be the level of the root */
For lev = 1 to L do
    For each vertex u on level lev do
        W(u,A) = AGGREGATE(W(x,i), t(x), W(y,i), t(y))
        W(u,P)=  PPARALLELIZE(W(x,i),t(x),c(x),W(y,j),t(y),c(y))
        If start time of W(u,A) < start time of W(u,P)
            then W(u) = W(u,A)
            else W(u) = W(u,P)
Return (W(r))
```

6.4 Extension to General Trees

The algorithms and procedures developed for binary trees are easily extended to arbitrary trees. To aggregate an arbitrarily long list of processes, the first two could be aggregated and the third merged with the result of the first two. A faster alternative would be to recursively merge pairs of workloads from the list until all are included.

Parallelization is simply the selection of a workload or set of workloads to be copied to the successor and determination of the start time. Selecting among many is essentially the same as selecting between two, and the start time will be the maximum among the possibilities.

The optimal, intractable solution is calculated by considering all parallelize/aggregate combinations, and pruning with lighter loads and single parallel decisions will provide good heuristics for reasonable results. A good greedy algorithm, WL-Schedule, uses a local optimal strategy where each vertex generates only one workload. Suppose that vertex u has k predecessors $u_1, u_2 \ldots u_k$. Let the starting time of $W(u_i)$ be s(i) and the task and communication costs be t(i) and c(i). Sort the workloads in non-increasing order by value of z(i)=s(i)+t(i)+c(i). The larger the value of z(i), the more the vertex should be aggregated with its neighbors since the start time of the aggregation does not include the communication time, c(i). The algorithm aggregates workloads in order of non-increasing z(i) until the resulting start time exceeds the z value of the next vertex. An additional processor is assigned to the next sequence of vertices. The aggregation repeats until the start time exceeds the next z value. The process repeats until all vertices are assigned. A conceptual diagram of the process is given in Figure 5.

```
Algorithm WL-Schedule(P, u, Root(u), W(u))
/* Input:  processor counter P for scheduling grains; vertex label u*/

/*Output: W(u); updated processor count P; schedules of grains decided at u*/

begin
Sort the predecessors of u in order of decreasing values of s(j)+t(j)+c(j) and place
on process list, PL.
If u is the root, set the finish time of the final schedule to s(u)+t(u).
Starting with first predecessor, select workloads in order from PL and aggregate
    into workload W(u) until the start time s(u) is larger than the value of z for
    the next member of PL.
while not all predecessors have been inspected
    Starting with next predecessor, select workloads in order from PL and aggregate
        until the start time is larger than the value of z for the next member of PL.
     increment P.
    assign aggregation to processor P.
/*end while*/
```

If only the finish time of the final schedule is required, the while loop is not necessary. The purpose of the while loop is to aggregate processes into grains and schedule the grains on processors. The while loop guarantees that the starting time of task u will not be delayed by these grains.

The entire tree is scheduled with a post-order traversal and calls to WL-schedule at each non-leaf vertex. The recursive algorithm, is called Tree-Schedule. To find the schedule for a tree DAG rooted at R, we call Tree-Schedule(1,R,true).

```
Algorithm Tree-Schedule(P, r, Root(r));
/*  Input:  the root r of a tree DAG, a predicate Root(r) which is true if r is the
    original root. */

/* Output: the schedule of the tree DAG */
begin
    if (r is a leaf) and Root(r) the schedule t(r) on processor 1 /* single task*/
    else if (r is leaf) then update workload (0,0)
    else
        begin
        for each predecessor r_i of r call Tree-Schedule(P,r_i,false);
        call WL-Schedule(P,r,Root(r),W(r))                    /*schedule r*/
        end
end
```

7. Conclusions

The workload schedule contains all the information necessary to make optimal decisions to parallelize or merge processors. It captures the essence of the communication/computation trade-off and allows the scheduling of processes during communication delay periods. It is a mechanism by which optimal schedules can be generated while considering execution and communication costs.

Through the ideas of workload scheduling we extended scheduling theory by proving that the parallel scheduling of programs with tree dependence structures is an NP-complete problem when arbitrary communication and execution costs are assigned and when the processes are scheduled on an unbounded number of processors. This result is in sharp contrast to the same problem with the additional restriction that communication costs are bounded by the execution costs. In this latter case, an optimal schedule can be found in O(n) time [PY90].

We also devised a set of algorithms for efficiently scheduling trees. The algorithms range from determining the optimal schedule, intractable in the general setting, to an efficient greedy algorithm with a sound heuristic basis. Testing is underway to measure the trade-off between the processing cost of optimal scheduling and the adequacy of heuristically created schedules.

Bibliography

[AHC90] F.D. Anger, J.J. Hwang, and Y.C. Chow, "Scheduling with Sufficient Loosely Coupled Processors," *Journal of Parallel and Distributed Comput.*, 9, 1990, pp. 87-92.

[Co76] E.G. Coffman Jr., (Ed.), *Computer and Job-Shop Scheduling Thoery*, John Wiley & Sons, New York, 1976.

[GJ79] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.

[GY90] A. Gerasoulis and T. Yang, "On the Granularity and Clustering of Directed Acyclic Task Graphs," *LCSR-TR-153, Dept. of Computer Science, Rutgers University*, Sept. 1990, pp. 1-28.

[GVY] A. Gerasoulis, S. Venugopal, and T. Yang, "Clustering Task Graphs for Message Passing Architectures, "*Proc. of ACM Int'l conf. on Supercomputing*, 1990, pp. 447-456.

[Gr69] R.L. Graham, "Bounds on Multiprocessing timing Anomalies, " *SIAM J. Appl. Math.*, Vol. 17, No. 2, Mar. 1969, pp. 416-429.

[GLLR79] R.L. Graham, E.L. Lawer, J.K. Lenstra, and A.H.G. Rinnooy Kan, "Optimization and Approximation in Deterministric Sequencing and Scheduling: A Survey," *Ann. Discrete Math.*, 5, 1979, pp. 287-326.

[HS84] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1984.

[Hu61] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Oper. Res.*, Vol. 9, No. 6, 1961, pp. 841-848.

[HCAL89] J.J. Hwang, Y.C. Chow, F.D. Anger, and B.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Comput.*, Vol. 18, No. 2, Apr. 1989, pp. 244-257.

[KB88] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Paralle Computation upon Multiporcessor Architectures," *Proc. of Int'l Conf. on Parallel Processing*, Vol. III, 1988, pp. 1-8.

[KL88] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing," I*EEE Softw.*, Jan. 1988, pp. 23-32.

[LHCA88] C.Y. Lee, J.J. Hwang, Y.C. Chow, and F.D. Anger, "Multiprocessor Scheduling with Interporcessor Communication Delays," *Oper. Res. Lett.*, 7, 3, 1988, pp. 141-147.

[Li90] Z.Liu, "A Note on Graham's Bound," *Inform. Processing Lett.*, 36, OCt. 1990, pp. 1-5.

[MG89] C. McCreary and H. Gill, "Automatic Determination of Grain Size for Efficient Parallel DProcessing," *Comm. of ACM*, Vol. 32, No. 9, Sept. 1989, pp. 1073-1078.

[MG90] C. McCreary and H. Gill, "Efficient Exploitation of Concurrency Using Graph Decomposition," *Proc. of Int'l Conf. on Parallel Processing*, Vol. II, 1990, pp. 199-203.

[PU87] C.H. Papadimitrou and J.D. Ullman, "A Communication-Time Tradeoff," *SIAM J. Comput.*, Vol. 16, No. 4, Aug. 19876, pp. 639-646.

[PY90] C.H. Papadimitrou and M. Yannakakis, "Towards an Architecture-Independent Analysis of Parelle Algorithms," *SIAM J. Comput.*, Vol. 19, No. 2, April 1990, pp. 322-328.

[RND77] E.M. Reingold, J. Nievergelt, and N.Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

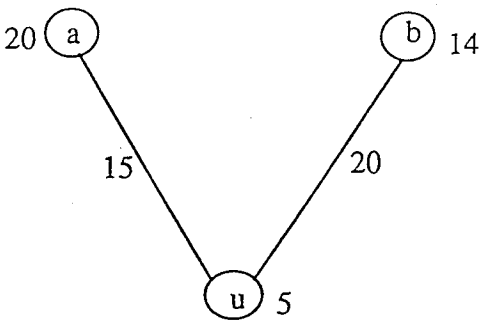
[RL90] H.E. Rewini and T.G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, 9, 1990, pp. 138-153.

[Sa89] V.Sarkar, *Partioning and Scheduling Parallel Programs for Execution on Multiprocessors*, The MIT Pross, Cambridge, Massachusetts, 1989.

[TDP89] A.E. Terrano, S.M. Dunn, and H.E. Peters, "Using an Architectural Knowledge Base to Generate Code for Parallel Computer," *Comm. of ACM,* Vol. 32, No. 9, Sept. 1989, pp. 1065-1072.

[WG90] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. on Parallel and Distributed Systems,* Vol. 1, No. 3, July 1990, pp. 330-343.

[Yu84] W.H. Yu, "LU Decomposition on a Multiprocessing System with Communication Delay," Ph.D. Theses, Dept. of EE and CS, University of California, Berkeley, 1984.



W(u,Pa) = ([0,20], 34)
W(u,Pb) = ([0,14], 35)
W(u,A) = ([0,34], 34)

Fig. 1 Workload example.

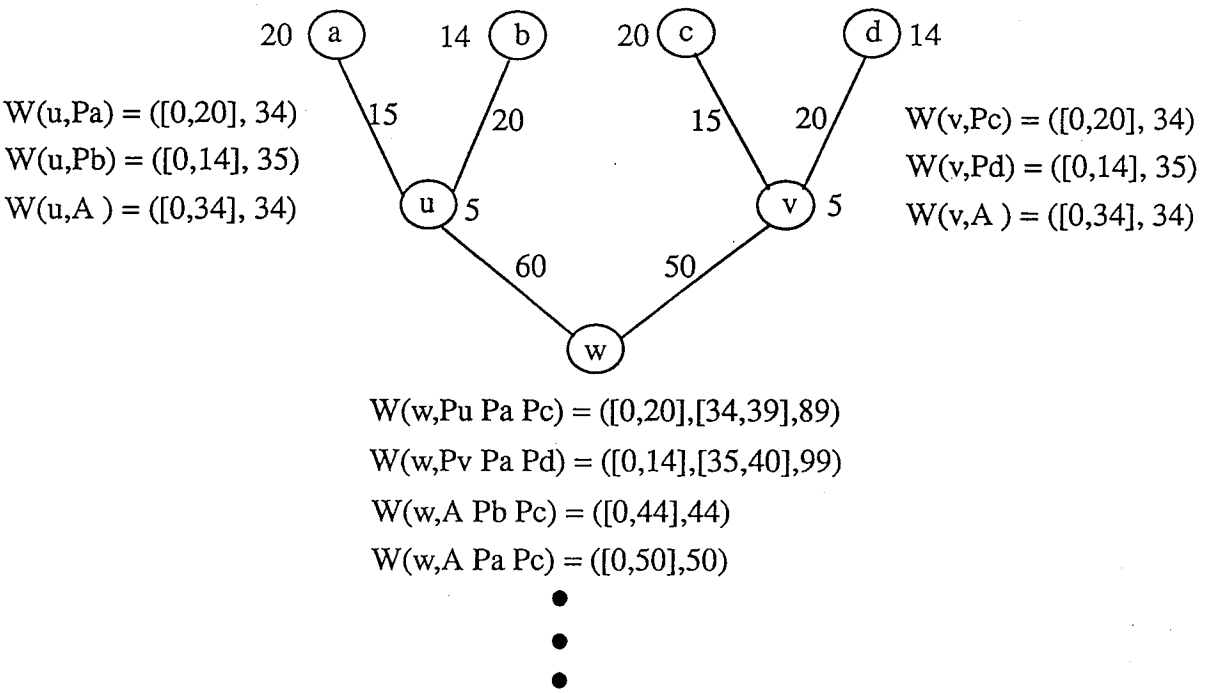


Fig.2 Sample workload schedules.

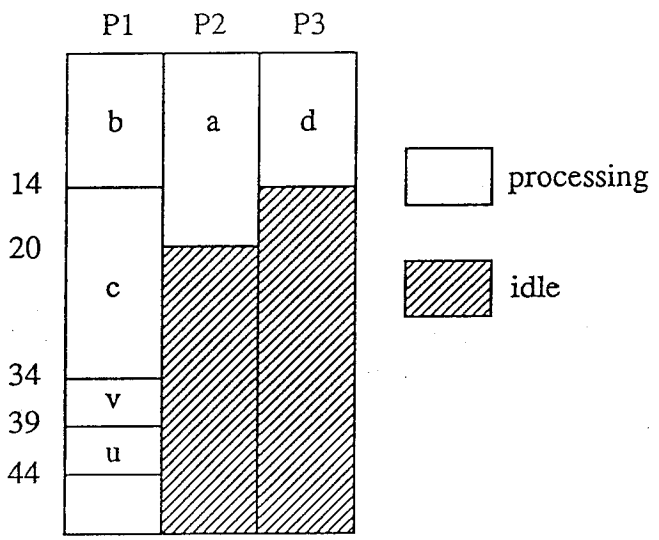
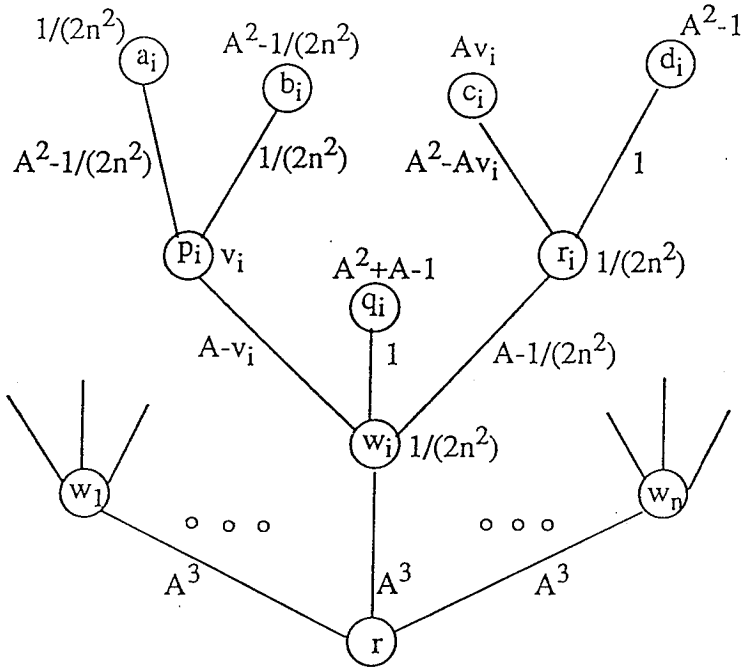


Figure 3. Optimal Schedule: W(w A Pb Pc) for Figure 2



$W(p_i) = ([0,1/(2n^2)],A^2)$

$W(r_i) = ([0,Av_i],A^2)$

$W(w_i,1) = ([0,1/(2n^2)],[A^2,A^2+v_i],A^2+A)$

$W(w_i,2) = ([0,Av_i],[A^2,A^2+1/(2n^2)],A^2+A)$

Fig.4 The construction for the NP-Complete proof.

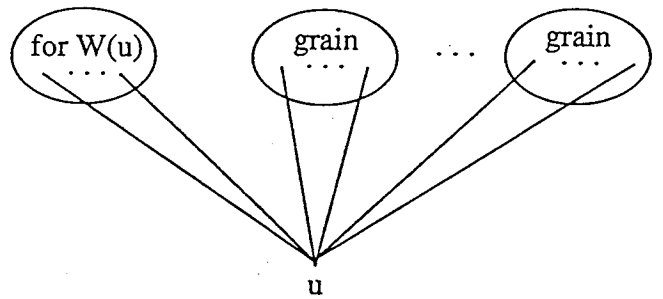


Figure 5. The scheduling of the algorithm WL-Schedule at node u.