WORKLOAD SCHEDULING FOR PARALLEL SYSTEMS

Technical Report CSE-91-16

Carolyn McCreary

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

Yahui Zhu

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

December 1991

Workload Scheduling for Parallel Systems

C. L. McCreary*

Department of Computer Science and Engineering,

Auburn University, AL 36849

Yahui Zhu

Department of Computer Science,

North Dakota State University,

Fargo, ND 58105-5075

* This research is sponsored by NSF grant no. CCR-8909253

List of Symbols - none

Workload Scheduling for Task Graphs with Communication Costs on Parallel Systems

Abstract

In this paper we extend the classical work on multiprocessor scheduling in four major ways:

1. A new notion of scheduling, called the *workload schedule* is defined. The workload captures the essence of the communication/computation trade-off, and it is a mechanism by which optimal schedules can be generated.

2. The theory of scheduling is extended by the statement and proof of a new NP-complete result. We prove 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 [18].

3. Several algorithms for scheduling trees are presented. These range from determining the optimal schedule to an efficient greedy algorithm with a sound heuristic basis. The optimal algorithm is intractible with running time $O(4^n)$ while the greedy algorithm runs in O(n) time.

4. The trade-off between generating optimal schedules and creating scheduling programs that execute in a reasonable amount of time is demonstrate through analysis of data output from seven programs. The analysis shows a "near-optimal" schedule is obtained by one algorithm with "near-greedy" running time.

1. Background

A sequential program is commonly represented as a directed acyclic graph (DAG) with node and edge weights[21.]. Each vertex in a DAG denotes a task and the weight, 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 [12], *clustering problem* in [11, 24], and *internalization pre-pass* in [21]. The problem is important because solution methods can be used to generate efficient parallel programs.

In classical DAG scheduling, communication costs are not considered [2, 7]. 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 time and degree of parallelism[17].

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 [4, 5, 11, 21, 23, 24]: Extending the critical path method due to Hu [9] 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 [5].
- List scheduling heuristics [1, 10, 12, 13, 14, 18, 20]: These algorithms assign priorities to the processes and schedule them according to a list priority scheme. Extending the list

scheduling heuristic due to [6] in classical scheduling, these algorithms use greedy heuristics and schedule tasks in a certain order. Task duplications have been used in [12,1, 20] to reduce the communication costs.

- Graph decomposition method [15, 16]: 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. Section 7 gives the results of empiracle testing, 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[18]. 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 [21], the Hypertool parallel programming environment [23], and the parallel code generation system in [22].

3. The Workload Concept

In workload scheduling a *workload schedule* is defined at each 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 vertex is executing some code, while the start time gives the time when the 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:

$$W(u) = ([a_1,b_1],[a_2,b_2], \ldots [a_k,b_k], s(u)), \qquad \text{where}$$

$$a_1 = 0,\ a_i <= b_i \text{ and } b_i < a_{i+1} \text{ for all i and } b_k <= s(u).$$

The processor executing u is busy from a_i to b_i. For interval i, a_i is the busy start time and b_i is the busy end time. In workload W(u), the wimte toexecute u is not included, but s(u) gives the earliest time u's processing can begin.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.

The concept of workload scheduling is to keep track of the workloads on the critical path to the root and to make decisions at each vertex of the path that will minimize the total elapsed time. In a binary tree, two sibling tasks can be scheduled in the following ways: *parallelize* them by executing them on two separate processors, or *aggregate* them by executing them both on one processor. Parallelization may include a communication delay while the results of the two processes are gathered by one of the processors, while aggregation implies slower sequential processing. Knowledge of the busy intervals implies knowledge of the idle time and attempts are then made to fill these gaps with real work.

For a tree node *u* with ancestors *a* and *b*, there are four choices for the execution of *u*:

(i). *Parallelize right:* Place *a* on same processor as *u*, and execute in parallel with *b*.

(ii). *Parallelize left:* Place *b* on same processor as *u*, and execute in parallel with *a*.

(iii). *Aggregate:* Aggregate *a*, *b*, and *u* and place all three on the same processor.

(iv). *Separate:* Place *u* on a processor different from *a* or *b*.

In *parallelize right*, case (i), the execution of *u* cannot begin unil both *a* has completed and *b*'s results are passed to *u*. The workload at u will show a busy interval from *a*'s start time to its end time. The workload start time s(*u*) will be maximum(endtime of a, end time of b +b's commuication cost).*Parallelize left*, case (ii) is identical. For *aggregate*, case (iii), the busy interval will include the sum of the processing times of *a* and *b*. For *separate*, case (iv), *u*'s start time will be the maximun processing plus communication time of either *a* or *b*. There is no busy interval generated. For figure 1, $W_k(u)$ gives the workload schedule for case k. 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 u is place on its own processor, that processor is never busy, but u cannot begin execution until both a and b have completed and sent the results to u.

Extend the above example to a tree of depth 3, as shown in Figure 2 and consider the possible schedules for vertex w. There are four choices: vertices u and v can be (i) executed in parallel with u placed on the same processor as w, (ii) executed in parallel with v placed on the same processor as w, (iii) aggregated and placed with w, or (iv) executed in parallel and placed on a different processor from u. Furthermore, since there are 16 $W(u)$,$W(v)$ workload pairs, there are 64 possible workloads at vertex w, one of each of four cases at each paired workload. To illustrate their construction, consider $W_1(w)$ which is derived by executing u and v in parallel using $W_1(u)$ and $W_1(v)$ and placing u on the same processor as w. The resulting workload includes $W_1(u)$, 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 ($s(v)+t(v)+c(v)$):

$$W_1(w)=([0,20][34,39],89)$$

Aggregation combines all workloads, utilizing idle times whenever possible. $W_3(w)$ is an aggregation of $W_2(u)$ and $W_1(v)$.

$$W_3(w)=([0,14],[14,34],[34.39],[39,44],44) = ([0,44],44)$$

The optimal schedule is shown in Figure 3.

It may seem unnecessary to include case (iv) since the start time of u will be later than in either case (i) or (ii). However, optimal decisions are not based on the best local decision, and only by including this case can the optimal schedule be obtained. Consider the tree in figure 6. The best schedule at node w comes from executing c and d in parallel and sending both results to v on a different processor. In particular, *separate* at node v gives the schedule: W(v) = ([0,0],28) and *aggregate* at node u gives the schedule W(u) = ([0,20],21). Updating each to include the processing requirements of v and u gives W'(v) = ([0,0],[28,29],29) and W'(u) = ([0,21],21). Since W'(v) requires such a little amount of processing, its aggregation with W'(u) adds only a small

interval. W(w) from this aggregation is ([0,21],[28,29],29), the best at w.

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 four available options: parallelize left, parallelize right, aggregate and separate. 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.

The above examples show how the communication-time trade-off has been captured 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.

4.1 Algorithms for Creating the Workload

Four algorithms are given for binary trees: UPDATE which incorporates the execution requirements of the current vertex; and AGGREGATE, PARALLELIZE, and SEPARATE which generate the workloads.

Update

Recall that W(u) does not include the processing requirements of u. UPDATE adds u's processing to W(u). Algorithm UPDATE takes as input a workload, W(u) and a task time for u, t(u). If the busy end time of the last interval is the same as the workload's start time, ($s(u) = b_k$), 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, $W_1(w)$ is updated $W_1(u)$ 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 is then aggregated with $W_1(v)$ the workload is ([0,50],50). On the other hand, placing u with b creates workload ([0,14],35) and merging with $W_1(v)$ gives workload $W_3(w)$ = ([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, in Figure 2, $W_1(u)$ assigns *a* to the same processor as *u*. $W_1(v)$ assigns *c* to the same processor as *v*, and the aggregation assigns all four of these vertices to the same processors as *w*. Meanwhile *b* and *d* are assigned to two different processors,and 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) = ([e_1,f_1], ..., [e_r,f_r], s(u)) */

UPDATE(W(x),t(x)); /* denote W(x) = ([a_1,b_1],[a_2,b_2], ...[a_p,b_p], 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.

Suppose W(x) = ([0,2],[3,5],10), t(x) = 2, W(y) = ([0,1],[7,8],[11,12],14) and t(y)=2. UPDATE W(x) and W(y) to W'(x) = ([0,2],[3,5],[10,12],12 and W'(y) = ([0,1],[7,8],[11,12],[14,16],16). Then merging [0,2],[0,1] and [3,5] gives [0,5]; merging [10,12] and [11,12] gives [10,13]. 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 PARALLELIZE algorithm copies two workloads, again yielding a linear time algorithm.

Separate

The procedure SEPARATE creates the workload for u that includes no work of u's ancestors and specifies the start time when u's ancestors have completed their work and communicated the necessary information to u.

```
Algorithm SEPARATE ( 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) and communication
requirements, c(x) and x(y).*/
/* Output:  the workload W(u) = ([0,0], s(u))
UPDATE( W(x),t(x)); UPDATE (W(y),t(y));
s(u) = max(s(x)+c(x), s(y)+ c(y))
W(u) = ([0,0],s(u))
Return W(u)
```

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, half as many 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 $W_1(u)$ = ([0,8], [10,15], 17)and $W_2(u)$ = ([0,4], [10,12],17). The updated workloads (including u's execution requirements) are $\overline{W}_1(u)$ = ([0,8], [10,15], [17,20], 20) and $\overline{W}_2(u)$ = ([0,4], [10,12],[17,20],20). Both have the same start time, but $\overline{W}_2(u)$ 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 $W_1(u)$ gives the workload ([0, 27],27), and W(v) aggregated with $W_2(u)$ 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, $W_1(x) = ([a_1,b_1],[a_2,b_2], \ldots [a_p,b_p], s_1(x))$ and $W_2(x) = ([c_1,d_1],[c_2,d_2], \ldots [c_q,d_q], s_2(x))$, then workload $W_1(x)$ is said to be a ***subschedule*** of $W_2(x)$ if

(i) 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

(ii) $s_1(x) \le s_2(x)$.

$W_1(x)$ is a *proper subschedule* of $W_2(x)$ 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 $W_1(x)$ is a subschedule of $W_2(x)$, $W_2(x)$ is said to be a *superschedule* of $W_1(x)$.

In the example above $W_2(x)$ is a subschedule of $W_1(x)$.

Lemma 1: Let $W_1(u)$ and $W_2(u)$ be two workloads at vertex u and let $W_1(w)$ and $W_2(w)$ denote the workloads obtained by aggregating $W_1(u)$ and $W_2(u)$ resp. with W(v). If $W_1(u)$ is a subschedule of $W_2(u)$ then $W_1(w)$ is a subschedule of $W_2(w)$.

Proof: Follows directly from the procedure AGGREGATE and the definition of subschedule.

Let $W_{ix}(w)$ denote the workload at w resulting from parallelizing $W_i(x)$ with $W(v)$ and placing vertex x on the processor with w.

Lemma 2: Let $W_{1u}(w)$ and $W_{1v}(w)$ be the workloads obtained by parallelizing $W_1(u)$ and W(v). $W_{2u}(w)$ and $W_{2v}(w)$ denote the workloads obtained by parallelizing $W_2(u)$ and W(v). If $W_1(u)$ is a subschedule of $W_2(u)$, then $W_{1u}(w)$ is a subschedule of $W_{2u}(w)$ and $W_{1v}(w)$ is a subschedule of $W_{2v}(w)$.

Proof: Follows directly from the procedure PARALLELIZE and the definition of subschedule.

Lemma 3: Let $W_1(u)$ and $W_2(u)$ be two workloads at vertex u and let $W_1(w)$ and $W_2(w)$ denote the workloads obtained by the algorithm SEPARATE with inputs ($W_1(u)$, W(v))and ($W_2(u)$,W(v)) resp. If $W_1(u)$ is a subschedule of $W_2(u)$ then $W_1(w)$ is a subschedule of $W_2(w)$.

Proof: Follows directly from the procedure SEPARATE and the definition of subschedule.

4.3 An Optimal Solution

At each vertex in the binary tree, exactly four choices are presented: aggregate, parallelize right, parallelize left, and separate. By lemmas 1, 2, and 3, workloads that are superschedules of the other workloads can be eliminated from consideration without losing options that may lead to an optimal schedule. 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 $W_i(x)$ and $W_j(y)$ 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_index(u)) = AGGREGATE(W_i(x), t(x), W_j(y), t(y))
                If W_index(u) is not a superschedule of any W_k(u), 1<=k<index
                    then increment index
                If any W_k(u) is a superschedule of W_index(u), delete it from the set
```

of workloads at u and decrement index

$W_{index}(u)), W_{index+1}(u)) =$

PARALLELIZE($W_i(x)$, t(x),c(x), $W_j(y)$, t(y),c(y))

If $W_{index}(u)$ is not a superschedule of $W_k(u)$, 1<=k<index

then temp = 1 else temp = 0;

If $W_{index+1}(u)$ is not a superschedule of $W_k(u)$, 1<=k<index

then index = index+1+temp else index = index + temp

If any $W_k(u)$ is a superschedule of $W_{index}(u)$ or $W_{index+1}(u)$,

delete it from the set of workloads at u and

decrement index

$W_{index}(u))$ =SEPARATE ($W_i(x)$, t(x),c(x), $W_j(y)$, t(y),c(y))

If $W_{index}(u)$ is not a superschedule of any $W_k(u)$, 1<=k<index

then increment index

If any $W_k(u)$ is a superschedule of $W_{index}(u)$, delete it from the set

of workloads at u and decrement index

n(u) = index

Among all n(u) workloads, $W_i(r)$, 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 4: 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, 2, and 3 imply that superschedules can never generate

an earlier start time. []

Theorem 5: For a complete binary tree of n nodes, there are $4^{\frac{(n-1)}{2}}$ workloads at the root.

Proof: Induction proves the result using the base case as a tree with 3 nodes and 4 workloads at the root.

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[3]. 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 [1] 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. Papadimitriou and Yannakakis [18] 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. The wild swings in complexity of similar problems emphasize the need to pay careful attention to the complexity of scheduling under new machine models.

We prove that 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 [1]. 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 [3] 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, $B = \{v_i: 1 \leq i \leq n, v_i \text{ a positive integer}\}$, 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 that corresponds to this instance of the partition problem, and 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 symbol >> represents a large value (at least the sum of all processing) and blocks from consideration using the communication as an option for optimal scheduling. 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 $(v_i,>>)$. Vertex w_i has two predecessors, denoted as p_i and r_i, with corresponding tc -pairs $(v_i-1,A-v_i+1)$ and $(1,A-1)$, respectively. Vertex p_i has two predecessors, a_i and b_i with tc-pairs $(1, >>)$ and $(A^2-1,1)$. Vertex r_i has two predecessors, c_i and d_i with their tc-pairs $(Av_i-1,>>)$ and $(A^2-1,1)$. Clearly the tree can be built in polynomial time.

The possible workloads at p_i include

$W_1(p_i) = ([0,A^2],A^2)$ (aggregation)

$W_2(p_i) = ([0,1],A^2)$ (parallelize and place p_i on same processor as a_i)

$W_3(p_i)$= ([0,A^2-1],>>) (parallelize; place p_i on same processor as b_i)

$W_4(p_i)$= ([0,0],>>) (place p_i on its own processor).

Because >> exceeds the total processing of all nodes, any schedule that includes its cannot be better than the serial execution, and can be eliminated from consideration. Of the remaining choices, $W_2(p_i)$ is the only one that is not eliminated by the subschedule relationship.

Similarly, the only feasible workload at r_i is $W(r_i)$ = ([0,Av_i-1],A^2) (parallelize and place r_i on same processor as c_i)

The possible workloads at w_i that are not superschedules include:

$W_1(w_i)$ = ([0,0],A^2+A) (separate w_i's processing)

$W_2(w_i)$ = ([0,Av_i],[A^2,A^2+v_i],A^2+v_i) (aggregate p_i and r_i and place on same processor as w_i)

The communication costs to the root are too large to benefit from executing any two vertices concurrently or executing the root on a separate processor. Aggregating all vertices w_i produces the optimal schedule. The only question remaining is which workload, W_1 or W_2 to choose at each node, w_i.

Let S be the subset of vertices $w_{i_1} \ldots w_{i_k}$ for which the workload $W_2(w_i)$ is chosen. Let |S| denote $\sum_{j=1}^{k} v_{i_j}$ corresponding to w_{i_j} in S. The work done by all $W_1(w_i)$ is then 2A -|S|.

If |S| < A, the workload at r is W(r) = ([0,A|S|],[A^2,A^2+ |S|],[A^2+A, A^2+3A-|S|],A^2+3A-|S|).

If |S| > A, W(r) = ([0,A|S|],[A|S|,A|S|+|S|],[A|S|+|S|, A|S|+2A], A|S|++2A).

If |S| = A, W(r) = ([0,A^2],[A^2, A^2+A],[A^2+A,A^2+2A],A^2+2A).

The first two terms of W(r) come from the busy intervals in updated $W_2(w_i)$; the third term from the busy interval in updated $W_1(w_i)$. The minimal start time comes when |S| = A.

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 time algorithm that results in an optimal schedule 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, $W_1(x) = ([a_1,b_1],[a_2,b_2], ...[a_p,b_p], s_1(x))$ and $W_2(x) = ([c_1,d_1],[c_2,d_2], ...[c_q,d_q], s_2(x))$, $W_1(x)$ is said to be ***lighter*** than $W_2(x)$ iff $\Sigma(b_i-a_i) \leq \Sigma(d_i-e_i)$ and $s_1(x) \leq s_2(x)$, i.e. the total busy time from $W_1(x) \leq$ the total busy time from $W_2(x)$ and the start time of $W(x,1)$ is less than the start time of $W_2(x)$. $W_1(x)$ is a ***strictly lighter*** load if $\Sigma(b_i-a_i) < \Sigma(d_i-e_i)$ and $s_1(x) \leq s_2(x)$, or $\Sigma(b_i-a_i) \leq \Sigma(d_i-e_i)$ and $s_1(x) < s_2(x)$. $W_2(x)$ is said to be ***heavier*** than $W_1(x)$.

For example, let $W_1(v) = ([0,4],[8,10],10)$ and $W_2(v) = ([0,3],[4,7],[9,10],10)$. $W_1(v)$ is lighter than $W_2(v)$ since the total busy times of $W_1(v)$ and $W_2(v)$ are 6 and 7 respectively. Notice that $W_1(v)$ is not a subschedule of $W_2(v)$.The subschedule relationship is stronger than the lighter relationship. Whenever $W_1(u)$ is a subschedule of $W_2(u)$, it is also true that $W_1(u)$ is lighter than $W_2(u)$.

A reasonable algorithm heuristic algorithm LIGHTER_LOAD applies this pruning to the OPTIMAL SCHEDULE algorithm by applying the test for lighter loads in place of the test for

superschedules.

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 Delete Separate Choice

The use of the choice SEPARATE for producing workloads is questionable. By removing that choice from the production of workloads at any node the number of workloads generated will be $O(3^n)$ rather than $O(4^n)$.

6.4 Local Greedy Decision

When scheduling overhead must be minimized, a greedy O(n) algorithm can be used. In this algorithm only one workload will be associated with any node, the workload with the smallest start time. Note that SEPARATE's start time is the maximum of the two parallel cases and will never be selected by the greedy algorithm.

```
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
        W1(u) = AGGREGATE(Wi(x), t(x), Wj(y), t(y))
        W2(u)= PPARALLELIZE(Wi(x),t(x),c(x),Wj(y),t(y),c(y))
        If start time of W1(u) < start time of W2(u)
            then W(u) = W1(u)
            else W(u) = W2(u)
Return (W(r))
```

6.5 Extension to General Trees

The algorithms and procedures developed for binary trees can be extended to arbitrary trees. 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. The algorithms to aggregate, parallelize and separate are essentially the same as for the binary case. For an arbitrarily long list of predecessors, aggregation of all the processes could proceed pairwise or recursively in a divide and conquer paradigm. Parallelization selects the predecessor workload with largest value $z(x) = s(x)+t(x)+c(x)$ and copies that workload. The new workload's start time is the maximum of the remaining predecessors values, $z(y)$.

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 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. Tests and Results

Seven scheduling algorithms based on workload scheduling were implemented. The first was the OPTIMAL SCHEDULE from section 4.3. The second algorithm, PAR-ONE chose only one parallel workload rather than including both as in the optimal algorithm i.e. it substituted algorithm PPARALLIZE for PARALLELIZE. The third, LIGHT, pruned by lighter loads as well as using only one parallel choice. The next three algorithms, OPT-SEP, PAR-ONE-SEP, and LIGHT-SEP were a repeat of the first three where the workloads generated by SEPARATE were not included. The final algorithm was the GREEDY algorithm described in section 6.4. Each algorithm was run with random input for both execution and communication times. Two measurements were calculated for each algorithm: EndTime, the end time of the fasted schedule generated and #Wklds, the number of workloads generated at the root node. The latter was calculated as an estimate of the amount of work an algorithm did.

Test one - communication and execution times are nearly equal

In the first study, the communication times were random variables of the same magnitude as the execution times. Test cases were generated for trees with 31 nodes. Each test case was repeated with 5 sets of random input. All tests were run on a PC with a 386 processor using Turbo Pascal. As expected, the OPTIMAL SCHEDULE produced the shortest schedule, but took the longest time and evaluated the greatest number of workloads at the root. The average of #Wklds for the optimal schedule was 33.6, not nearly the $4^{15} = 2^{30}$ that would have been produced without eliminating subschedules.

The remaining algorithms produced excellent schedules in this test. None produced a schedule where EndTime was greater than 6.5% longer than the optimal. #Wklds varied as expected, and

the algorithms fell into two groups: High value of #Wklds included OPT, PAR-ONE and OPT-SEP. For these algorithms the average #Wklds exceeded 20, and for the others, #Wklds was no greater than 4.

GREEDY, as expected, produced the longest schedule and took the shortest amount of time to execute, having only one workload to generate at each node. For this experiment, GREEDY'S schedule was at most 6.5% higher than the optimal (229 vs. 215).

Test two - communication times greater than execution times

The good values of EndTime of the first test should have been anticipated because Anger, Hwang and Chow[1] prove that when execution times are no greater than communication times, a greedy algorithm is optimal. A second test was run where the communication costs were larger than execution times. For this test the relative merits of the algorithms were more apparent.

Complete trees of 31 nodes were built, and random values in the range 10 - 50 were assigned as node execution times. Additional random values were generated for the communication times. Three ratios of communication time to execution time generated: communication was 2, 5, and 10 times execution. For one case where the communication time range was 10 times the execution time range, the greedy algorithm produced a schedule whose EndTime was 65% longer than optimal. The other results are summarized in Figures 7 and 8.

As in the first test, the algorithms grouped themselves into two classes: large #Wklds and low #Wklds. The most interesting observations are:

1. The SEPARTE choice often affects the schedule, but usually not by much (<5%) for random execution and communication times. Figure 6 gives an example where the use of SEPARATE makes a major impact on the schedule.

2. For larger communication/execution ratios up to the point where all algorithms choose the aggregate option, the discrepancies between the algorithms grows. Greedy produces worse schedules; OPTIMAL generates more workloads (and runs out of stack space in our implementation).

3. Although examples exist where choosing only one of the parallel options can be less than optimal (see Section 4.1 - parallelize) none of our runs found this to be the case. PAR-ONE always gave the optimal EndTime. (#Wklds was very large, however, and in almost as many cases as OPTIMAL, stack size was exceeded.)

4. LIGHT seems to be the most reasonable algorithm. In all cases we tested, it never generated more than 10 workloads at the root and always produced results that were within 7% of optimal. LIGHT creates a "near-optimal" schedule in "near-greedy" running time.

5. Despite the disparity between the algorithms, even GREEDY workload scheduling gives excellent results. For the example in Figure 6, load balancing gives a time of 67 compared to GREEDY's 45[15.].

Of the five algorithms between the one that produces the optimal schedule, OPTIMAL, and the one that is the fastest, GREEDY, LIGHT is the one that produces the best schedules for the amount of work it does. Figures 7 and 8 compare the results of these three algorithms. Figure 7 shows the average ratio of EndTime to optimal EndTime. For OPTIMAL this value is 1. Figure 8 gives the average #Wklds and for GREEDY this value is always 1. LIGHT is small for all cases in both figures.

8. 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 [18].

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 measured the trade-off between the processing cost of optimal scheduling and the adequacy of heuristically created schedules. The analysis shows a "near-optimal" schedule is obtained by one algorithm with "near-greedy" running time.

Acknowledgements: We wish to thank Kim Smith and Jeanne Stone for their work programming the algorithms and gathering the data.

Bibliography

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

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

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

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

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

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

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

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

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

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

11. Kim, S.J. and Browne, J.C. 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.

12. Kruatrachue, B. and Lewis, T. Grain Size Determination for Parallel Processing. *IEEE Softw.*, Jan. 1988, pp. 23-32.

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

14. Liu, Z. A Note on Graham's Bound. *Inform. Processing Lett.*, 36, Oct. 1990, pp. 1-5.

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

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

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

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

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

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

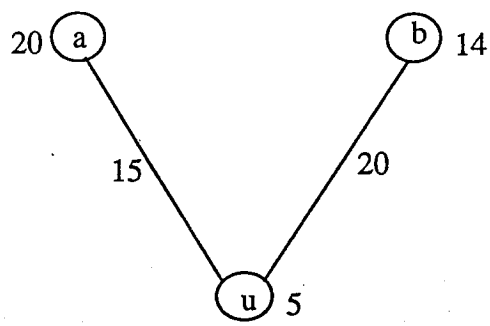
21. Sarkar, V., *Partioning and Scheduling Parallel Programs for Execution on Multiprocessors*. The MIT Pross, Cambridge, Massachusetts, 1989.

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

1065-1072.

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

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



$W_1(u) = ([0,20], 34)$

$W_2(u) = ([0,14], 35)$

$W_3(u) = ([0,34], 34)$

$W_4(u) = ([0,0], 35)$

Figure 1 Workload example.

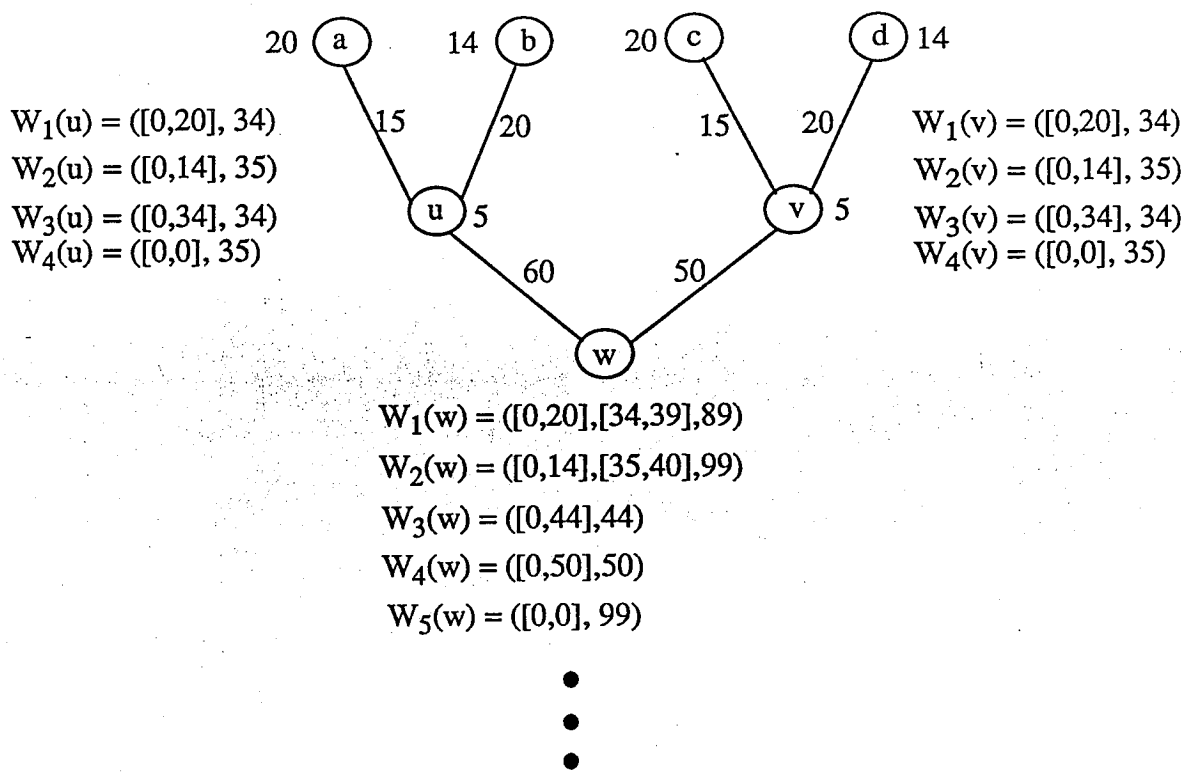


Figure 2. Sample workload schedules.

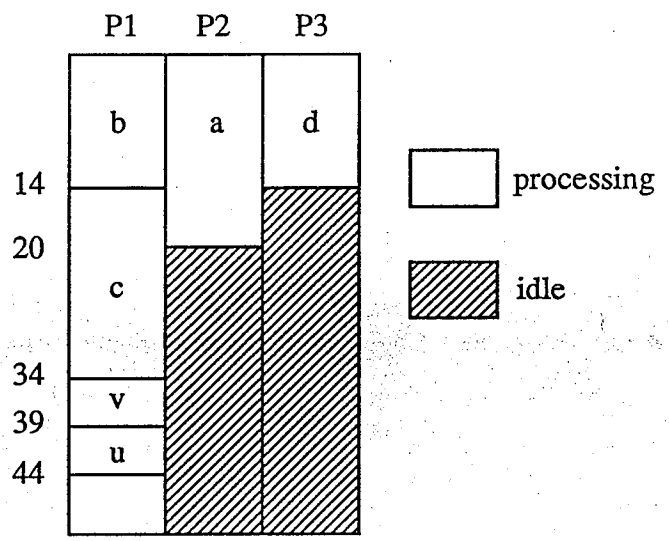
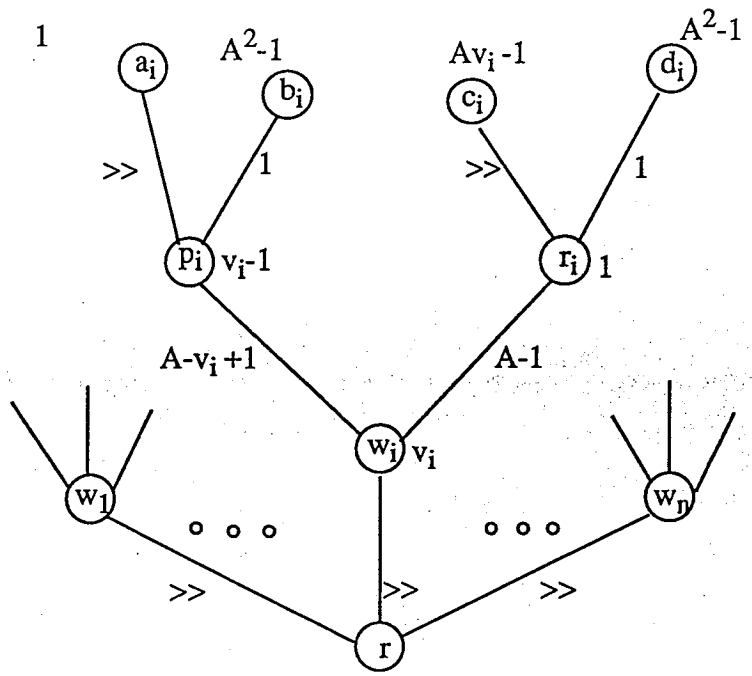


Figure 3. Optimal Schedule: $W_3(w)$ for Figure 2



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

$W(r_i) = ([0,Av_i\text{-}1],A^2)$

$W_1(w_i) = ([0,0],A^2+A)$

$W_2(w_i) = ([0,Av_i],[A^2,A^2+v_i],A^2+v_i)$

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

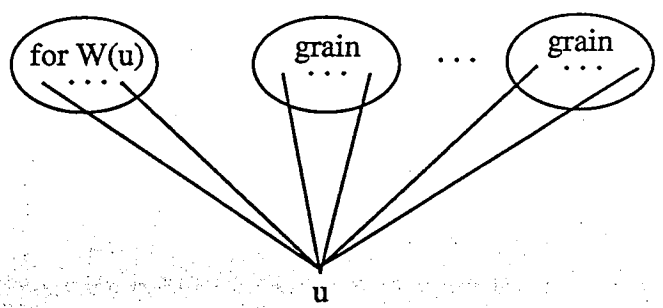


Figure 5. The scheduling of the algorithm WL-Schedule at node u.

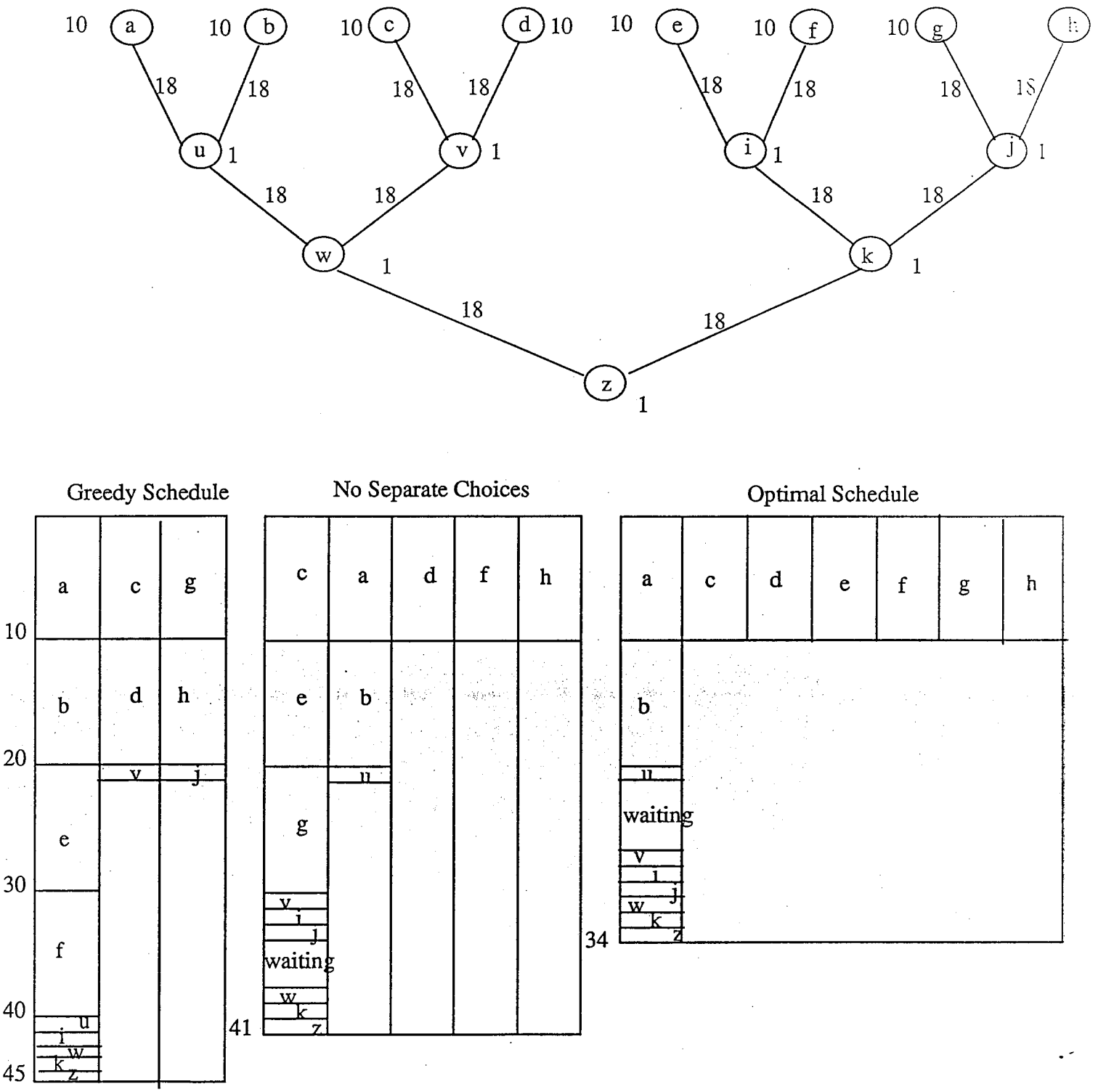


Figure 6. Comparison of Methods

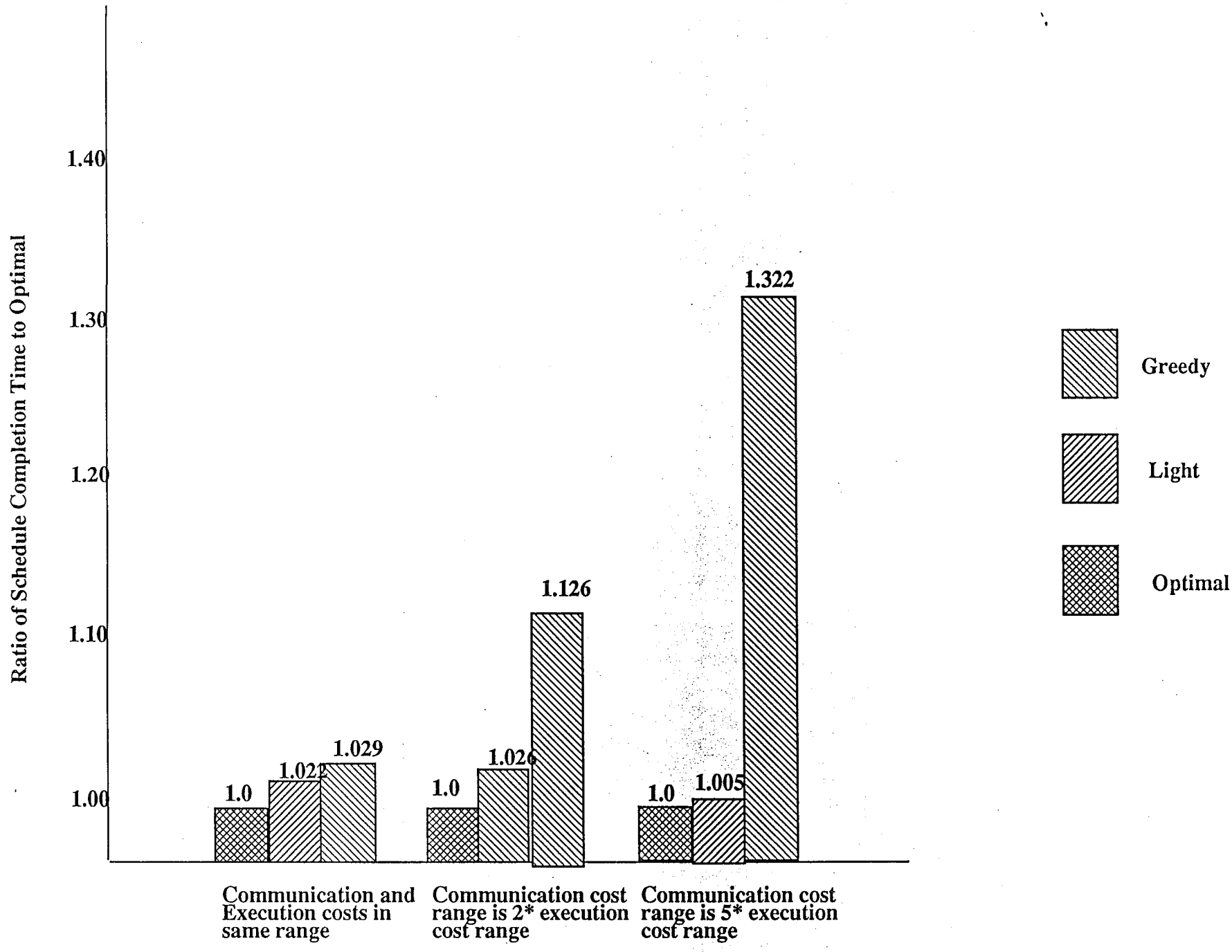


Figure 7. Ratio of Schedule Completion Time to Optimal

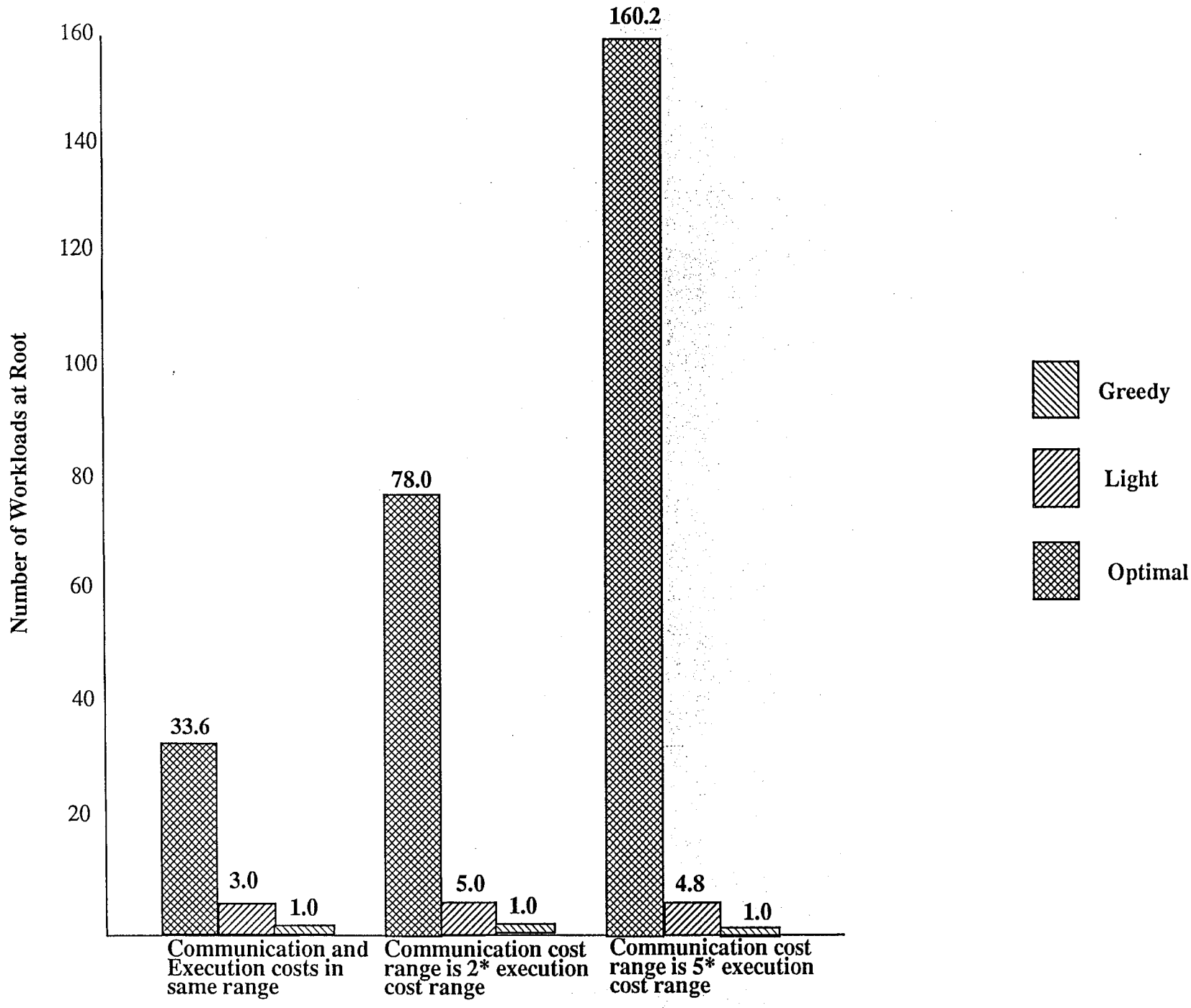


Figure 8. Number of Workloads at Root