Applications of UET Scheduling Theory to the Implementation of Declarative Languages

F. W. BURTON,*‡ G. P. McKEOWN†§ AND V. J. RAYWARD-SMITH†

* *School of Computing Sciences, Simon Fraser University, Burnaby, British Columbia, Canada, V5A 1S6*

† *School of Information Systems, University of East Anglia, Norwich NR4 7TJ.*

Motivated by our interest in the use of functional programming languages to write parallel programs, we have been led to consider the anomaly of more processors possibly leading to a slower execution time. After reviewing known results from the theory of list-scheduling, we introduce generalisations of the familiar breadth-first and depth-first tree-searching algorithms to arbitrary dags and consider them in a list-scheduling framework. We prove that with UET actions (corresponding to pre-emptive scheduling with integer execution times for actions), breadth-first scheduling never leads to an increase in the execution time when the number of processors is increased. We also prove that for any list-scheduling algorithm with UET actions, there is no speed-up anomaly when going from two to three processors.

Received April 1986, revised April 1987

1. INTRODUCTION

We are interested in highly parallel algorithms which can be run on an arbitrary number of processors. It is often relatively easy to analyse the expected behaviour of a program on a single processor, and sometimes the hypothetical behaviour of the program on an unbounded number of processors can also be analysed. In general, however, it is very difficult to analyse the behaviour of a program on *n* processors, for arbitrary *n*, particularly when dynamic process creation is supported by dynamic process scheduling.

One might expect to be able to get whatever speed is required (up to the speed of the system of unbounded size) by increasing the size of the system. However, there are many reports in the literature of parallel algorithms which require more time when the number of processors is increased (see, for example, Lai and Sahni).[16] In many cases, this is because the algorithm is non-deterministic and happens to make a good choice (or set of choices) on a particular number of processors and a poor choice (or set of choices) on a larger number of processors. However, even deterministic algorithms sometimes exhibit this behaviour.[10,11]

Clearly, an optimal scheduler can always do at least as well with more processors. (If all else fails, it can ignore the additional processors.) However, there are two important problems which arise when we attempt to use an optimal scheduler. First, we usually do not know how much time an action requires until it has terminated. In fact, in many cases, we don't even know that an action will exist until it can start. Second, even if we have perfect information, optimal scheduling is NP-hard.[23,24]

We are interested in scheduling policies which guarantee that increasing the number of processors will not slow the speed of the system, and in bounding the damage that can be done by increasing the size of the system with other scheduling algorithms. We will consider only deterministic algorithms where the total amount of work performed is independent of the size of the system. (That is, we exclude speculative algorithms,[1,2] as well as non-deterministic algorithms.)

We are particularly interested in the use of declarative programming languages (logic languages and functional languages) to write parallel programs. In such languages, parallelism is implicit. For example, consider the following divide-and-conquer algorithm to compute the factorial function.

```
factorial(n) = = product(1, n)
where
product(i,j) = =
  if i = j then i
    else let mid = = (i+j)/2 in
      product(i, mid) * product(mid + 1, j)
```

This simple functional program computes the product of integers in an interval by calculating the product of the integers in the left half of the interval and multiplying the result by the product of the integers in the right half. The two operands of '∗' in the bottom line can be evaluated in parallel. If this is done in every case, the result is a parallel algorithm that will run in $0(\log n)$ time, provided that enough processors exist to realise the potential parallelism.

With this approach to parallelism, there is often a very high potential for parallelism. An implementation must be responsible for scheduling tasks for execution. This approach has the following advantages.

(1) The result produced by a program does not depend on the number of processors. The programmer does not have to worry about timing errors, synchronising processes, or non-determinism. A correct program remains correct.

(2) The program can run on systems of differing size. It is not necessary to know the size of the system that will run the program when it is written. Similarly, upgrading to a larger system will not require existing programs to be modified.

However, a challenge to this approach to parallelism is in scheduling tasks in a reasonable manner with incomplete information. Not only will a system be

‡ The work of this author was supported by the National Science Foundation under grant no. ECS-8312748 and grant no. DMC-8514946.

§ To whom correspondence should be addressed.

unsure how much time a process will require, it will not know how many processes will come into existence until after much of the scheduling has been done.

Our approach is equally suitable for procedural programming languages which support dynamic task creation and leave the scheduling of tasks to the implementation.

We will regard a parallel program as a collection of actions with a partial order, $<$, defined on them. If a_i and a_j are actions and $a_i < a_j$ then a_i must terminate before action a_j may start. If two actions are unordered, then they may be performed in either order or in parallel. In scheduling theory, actions are often called tasks. However, in programming languages tasks are usually of coarser grain than what we have in mind. For example, a task may contain semaphore *wait* and *signal* operations. We would regard the work between any pair of synchronisation steps (including task creation and deletion as well as *waits* and *signals*, or similar) as an action.

Since we regard a parallel program as a partially ordered set of actions, we may represent such a program as a directed acyclic graph (dag). For example, let us suppose that the divide-and-conquer algorithm given above is to be used to compute 8! The program comprises three types of action: testing, parameter evaluation and combining. Let t_{ij} denote the testing action that compares the values of i and j, let e_{ik} and e_{kj} denote the actions that evaluate the parameter k in the function references *product*(i,k) and *product*(k,j), respectively, and let c_{ij} denote the multiplication of the values of *product*(i, mid) and *product*$(mid+1,j)$ to give the value of *product*(i,j). The dag representing the parallel program for computing *product*$(1,8)$ is then as given in Fig. 1.

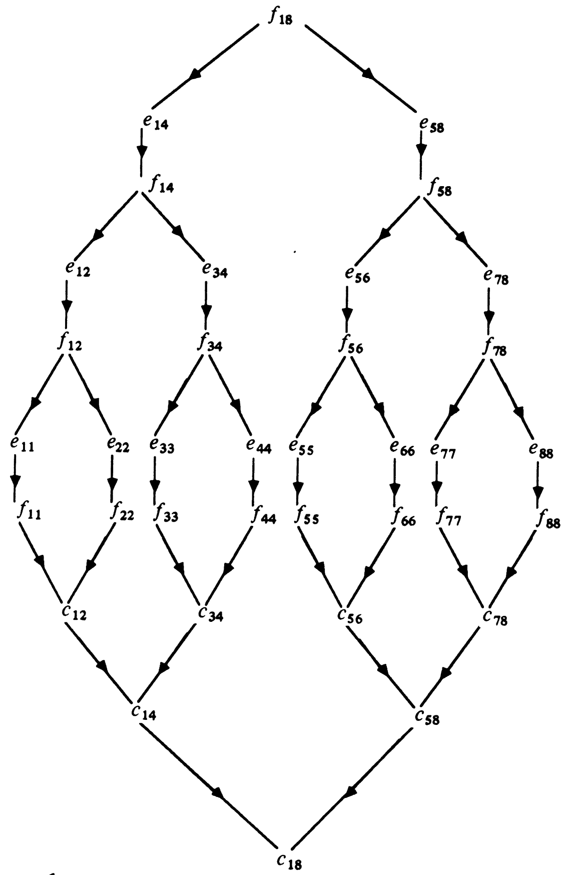


Figure 1.

2. BACKGROUND SCHEDULING THEORY

In this section we give a brief review of the relevant results from scheduling theory. We could not find a proof of Theorem 2 in the literature and the one given is our own. In Section 3 some new results of significance in the implementation of functional programs are given.

Let A denote a finite set of actions partially ordered by $<$. Each $a_i \in A$ is assumed to have execution time τ_i and we are interested in scheduling these actions on $m > 1$ identical processors, $P_1, P_2, \ldots, P_m$. Given any total order $\sqsubset$ containing $<$, there is an obvious associated algorithm for scheduling the actions. At any time a processor becomes free it is allocated the next unallocated action in the total ordering defined by $\sqsubset$. Such a schedule is called a *list schedule* and has been well studied in various texts and research papers in scheduling theory.[5,7]

For any set A, partial ordering $<$ and time function τ, we define

ω_0 to be the shortest length of any schedule (pre-emption allowed),

ω_0^n to be the shortest length of any non-pre-emptive schedule, and

ω_0^L to be the shortest length of any list schedule.

It is clear that

$$\omega_0 \leqslant \omega_0^n \leqslant \omega_0^L. \tag{1}$$

The question then arises as to how badly the best non-pre-emptive schedule can perform when compared with the best pre-emptive schedule, and how the best list schedule compares with both of these. In Ref. 22 it is shown that

$$\omega_0^L \leqslant \left(2-\frac{1}{m}\right)\omega_0, \tag{2}$$

and that this bound is achievable in the limit.

From (1) and (2) we can deduce that

$$\omega_0^L \leqslant \left(2-\frac{1}{m}\right)\omega_0^n \tag{3}$$

and again this bound is tight.

In Ref. 18 it is shown that

$$\omega_0^n \leqslant \left(2-\frac{2}{m+1}\right)\omega_0, \tag{4}$$

and it is easy to see that this bound is the best attained: consider a set of $m+1$ identical actions each of length m. The shortest non-pre-emptive schedule is of length $2m$, while the shortest pre-emptive schedule is of length $m+1$.

The next question that arises is: how does an arbitrary list schedule compare with the optimal list schedule? To answer this we need the following theorem, proved by Graham in 1966[10] (see also Refs 11 and 12).

Theorem 1

Let A be a set of actions executed twice. The first time we assume each action $a_i \in A$ takes τ_i time units, that these actions are partially ordered by $<$ and totally ordered by $\sqsubset$ containing $<$ and that we executed these actions on m identical processors. The second time we assume each a_i takes $\tau_i' \leqslant \tau_i$ time units, that the actions are partially ordered by $<'$ contained in $<$ and totally ordered by $\sqsubset'$ containing $<'$ and that they are executed on a system of m' identical processors. Using list scheduling, let ω

denote the finishing time in the first case and ω' in the second case. Then,

$$\omega' \leqslant \left(1+\frac{m-1}{m'}\right)\omega. \tag{5}$$

Not only is this bound tight but it can be achieved (asymptotically) by varying any one of the parameters of the theorem.

As an immediate corollary of this theorem, we see that for any fixed A, $<$, τ, if ω is the finishing time of a list-scheduling algorithm using an arbitrary $\sqsubset$ containing $<$, and ω_0^L is the length of the optimal list-scheduling algorithm, then

$$\omega \leqslant \left(2-\frac{1}{m}\right)\omega_0^L.^{10,11} \tag{6}$$

From the above results we can see that the length of an arbitrary list schedule is asymptotically no more than a small multiple of the optimal length. Finding the optimal (pre-emptive or non-pre-emptive) schedule is NP-hard[15] and so too is finding the best of all possible list schedules.[23] Hence there has been considerable interest in finding reasonable heuristics for list scheduling and for investigating those special cases for which polynomial time algorithms do indeed exist. Muntz and Coffman[19, 20] have proposed an heuristic for pre-emptive scheduling which is optimal for two processors and an arbitrary dag or when the dag is a forest and the number of processors is artibrarily chosen. If ω_{MC} denotes the length of a schedule computed by Muntz and Coffman's algorithm then

$$\omega_{\mathrm{MC}} \leqslant \left(2-\frac{2}{m}\right)\omega_0. \tag{7}$$

Moreover, examples exist to show that the bound is achievable asymptotically.[17] Muntz and Coffman's algorithm has also been further investigated for use on processors of different speeds in Ref. 13. A large number of heuristics for constructing list schedules are compared on a variety of test data in Ref. 9. As one would expect, successful algorithms tend to give preference to actions with many descendants and/or generations of descendants.

If we wish to implement a pre-emptive schedule using a list-scheduling approach we must allow activities on the dag to be regarded as a sequence of smaller activities allocated by the list scheduler. To model this situation, we consider dags in which all the activities have unit execution time (UET). UET systems have provided many of the most interesting results in scheduling theory. The UET list scheduler is simply described as follows.

```
{UET list scheduler}
{Actions are numbered so that $a_1 \sqsubset a_2 \sqsubset a_3 \ldots \sqsubset a_n$}
{initialize}
$U := A$; {$U$ is the set of unprocessed tasks}
$t := 0$; {$t$ is the clock}
while $U \neq \emptyset$ do
$R := \{a \in U \mid \nexists a' \in U$ s.t. $a' < a\}$;
{$R$ is the set of tasks which are available for processing
at time $t$}
$i := 1$;
  while $R \neq \emptyset$ and $i \leqslant m$
    do
    $l := \min\{i \mid a_i \in R\}$;
    allocate to processor $P_i$ during time interval $t$ the
      task $a_l$;
    $R := R - \{a_l\}$;
    $U := U - \{a_l\}$;
    $i := i+1$
  endwhile;
  $t := t+1$
endwhile
```

For any set of UET activities partially ordered by $<$ there is always some total ordering $\sqsubset$ containing $<$ such that the corresponding list scheduler gives an optimal schedule. The proof of this is relatively straightforward. First, consider any optimal schedule of the activities, say it takes total time ω_0. If this schedule has some processor, P_i, idling during the tth time interval and there is an activity $a_j \in A$ scheduled for a later time interval which could have been processed at the earlier time, then we can amend the schedule such that P_i processes a_j at time t. Doing this cannot increase the total time of the schedule (nor can it reduce it, since the original schedule was optimal). We continue applying similar amendments until we can assert that for all $1 \leqslant t < \omega_0$, there is a processor idling during time interval t only if all activities scheduled after time interval t have an ancestor scheduled during time interval t. Finally, we move activities scheduled during each time interval t from one processor to another so that if there is any idle time it is found on all processors $P_k, P_{k+1}, \ldots, P_m$ for some $k > 1$. We now define a total ordering on A as follows.

$a_i \sqsubset a_j$ iff in the final amended schedule, either a_i is processed before a_j or they are processed at the same time but a_i is processed on P_k and a_j on P_l, where $k < l$.

If the UET list scheduler is used under this total ordering, the resultant schedule must be the same as the amended schedule and hence have total time ω_0. We have thus established that

$$\omega_0^L = \omega_0 \quad \text{for UET systems.} \tag{8}$$

Further, (6) still holds and hence we can deduce that for a list schedule

$$\omega \leqslant \left(2-\frac{1}{m}\right)\omega_0 \quad \text{for UET systems.} \tag{9}$$

As we shall be seeing, examples exist to show that this worst-case ratio is achievable. Moreover, the problem of finding the optimal list schedule remains NP-hard even for UET systems (e.g. Ref. 24) and so once again we are led to consider heuristics.

The *level* of an activity, a, is defined to be the length of a longest path from a to a terminal activity. Hu proposes a total ordering where $a_i \sqsubset a_j$ if a_i has greater level than a_j (if the levels are the same, order arbitrarily).[14] This is optimal for forests but not for arbitrary dags. Denoting the length of a schedule constructed using this algorithm by ω_{HU}, Chen[3] and Chen and Liu[4] establish the following bounds for arbitrary dags:

$$\omega_{\mathrm{HU}} \leqslant \begin{cases} \frac{4}{3}.\omega_0 & \text{if } m = 2, \\ \left(2-\frac{1}{m-1}\right)\omega_0, & \text{otherwise.} \end{cases} \tag{10}$$

Coffman and Graham have developed this algorithm giving rules to decide whether $a_i \sqsubset a_j$ or not when a_i and

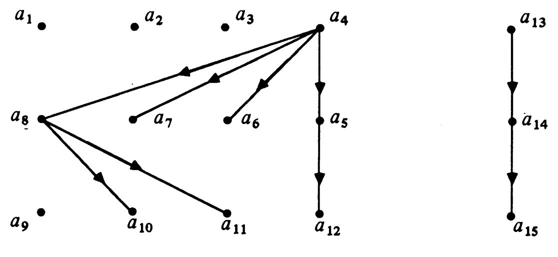
a_j have the same level.[6] As a result of this, their algorithm is optimal for $m = 2$ and $<$ being a forest. The previously cited Muntz and Coffman algorithm is a generalisation of this approach. The worst-case bound of the Coffman–Graham algorithm for an arbitrary dag, established in Ref. 17, is

$$\omega_{CG} \leqslant \left(2 - \frac{2}{m}\right)\omega_0. \tag{11}$$

Examples are given to show that this bound is achievable asymptotically. This algorithm is a list-scheduling algorithm; polynomial-time, non-list scheduling algorithms also exist which are optimal for two processors.[8]

One might expect that any of the following 'improvements' would reduce the length of schedule: (i) relaxing the partial order, $<$; (ii) reducing the time required to process some of the activities; (iii) increasing the number of processors. However, this is not the case – any of these three occurrences can actually *increase* the total execution time (see Ref. 11 or Ref. 12 for an illustration). Such behaviour is called an *anomaly*.

For example, it is easy to construct examples for the list scheduling of UET tasks where an anomaly arises when the number of processors is increased (see Fig. 2). It is even possible to get anomalous behaviour when the dag is a tree. In Fig. 2 we define $a_i \sqsubset a_j$ iff $i \leqslant j$ and use the UET list scheduler.



(*a*) The dag of activities.

	1	2	3	4	5
P_1	a_1	a_4	a_5	a_8	a_{10}
P_2	a_2	a_9	a_6	a_{12}	a_{11}
P_3	a_3	a_{13}	a_7	a_{14}	a_{15}

(*b*) Gantt chart for three processors.

	1	2	3	4	5	6
P_1	a_1	a_5	a_9	a_{13}	a_{14}	a_{15}
P_2	a_2	a_6	a_{10}	▨	▨	▨
P_3	a_3	a_7	a_{11}	▨	▨	▨
P_4	a_4	a_8	a_{12}	▨	▨	▨

(*c*) Gantt chart for four processors.

Figure 2. Increasing from three to four processors increases execution time.

Fig. 2 shows that for an arbitrary total ordering, $\sqsubset$, and an arbitrary partially ordered set, increasing the number of processors from three to four can increase the total execution time. The example can easily be generalised to produce an example to show that an increase from m processors to $m+1$ processors ($m \geqslant 3$) can produce a similar anomaly. However, an increase from two to three processors can never produce such an anomaly. To prove this we need the following definitions.

Let $B \subset A$ be a subset of A which inherits the partial order, $<$, and the total order, $\sqsubset$, defined on A. The time taken to schedule B using the UET scheduler on m processors is denoted by $\omega_m(B)$; ω_m denotes $\omega_m(A)$. The *ready set* of $B, r(B) = \{a \mid a \in B \wedge \nexists b \in B \text{ s.t. } b < a\}$ denotes that subset of B that can be scheduled in the first time interval. The *selected set of B using m processors*, $s_m(B)$ is that subset of $r(B)$ which is actually scheduled in the first time interval by the UET scheduler. Thus, if $|r(B)| \leqslant m$ $s_m(B) = r(B)$, but otherwise $s_m(B)$ is the m smallest elements of $r(B)$ under the ordering $\sqsubset$. Then the following is immediate from these definitions.

Lemma 1

If $B \neq \emptyset$ then $\omega_m(B) = \omega_m(B - s_m(B)) + 1$.

A second lemma we require is

Lemma 2

If $B' = B - \{a\}$ for some $a \in r(B)$ then $\omega_2(B') \leqslant \omega_2(B) \leqslant \omega_2(B') + 1$.

Proof

We proceed by induction on $|B'|$, the result clearly being true for $|B'| = 0$ or $|B'| = 1$. So, assume $|B'| \geqslant 2$ and the result holds for all sets of cardinality $< |B|$.

Case 1. $a \notin s_2(B)$. Then $s_2(B) = \{b, c\}$ where $b \sqsubset a$ and $c \sqsubset a$. It follows that $s_2(B') = \{b, c\}$ so, by Lemma 1, $\omega_2(B) = \omega_2(B - \{b, c\}) + 1$ and $\omega_2(B') = \omega_2(B' - \{b, c\}) + 1$. By the induction hypothesis, we can deduce that $\omega_2(B' - \{b, c\}) \leqslant \omega_2(B - \{b, c\}) \leqslant \omega_2(B' - \{b, c\}) + 1$ and hence, by adding 1 to each expression in this inequality that $\omega_2(B') \leqslant \omega_2(B) \leqslant \omega_2(B') + 1$.

Case 2. $s_2(B) = \{a\}$. Then $\omega_2(B) = \omega_2(B - \{a\}) + 1 = \omega_2(B') + 1$.

Case 3. $s_2(B) = \{a, b\}$ and $b \in s_2(B')$. If $s_2(B') = \{b\}$ then

$$\omega_2(B) = \omega_2(B - \{a, b\}) + 1 = \omega_2(B' - \{b\}) + 1 = \omega_2(B').$$

Otherwise, $s_2(B') = \{b, c\}$ where $c \in r(B')$ and hence $c \in r(B - \{a, b\})$.

Now $\omega_2(B') = \omega_2(B' - \{b, c\}) + 1 = \omega_2(B - s_2(B) - \{c\}) + 1$.

But by the induction hypothesis, $\omega_2(B - s_2(B) - \{c\}) \leqslant \omega_2(B - s_2(B)) \leqslant \omega_2(B - s_2(B) - \{c\}) + 1$. Hence, by adding 1 to each expression in this inequality, we get $\omega_2(B') \leqslant \omega_2(B) \leqslant \omega_2(B') + 1$.

Case 4. $s_2(B) = \{a, b\}$ and $b \notin s_2(B')$. In this case, $s_2(B') = \{c, d\}$ where $c \sqsubset b$ and $d \sqsubset b$. Moreover, $s_2(B - \{a, b\}) = \{c, d\}$. Now, let $B'' = B' - \{c, d\} = B - \{a, c, d\}$, then $\omega_2(B') = \omega_2(B' - \{c, d\}) + 1 = \omega_2(B'') + 1$ and $\omega_2(B) = \omega_2(B - \{a, b\}) + 1 = \omega_2(B' - \{b\}) + 1 = \omega_2(B' - \{b, c, d\}) + 2 = \omega_2(B'' - \{b\}) + 2$. By the induction hypothesis, $\omega_2(B'' - \{b\}) \leqslant \omega_2(B'') \leqslant \omega_2(B'' - \{b\}) + 1$. Therefore,

$$\omega_2(B) = \omega_2(B'' - \{b\}) + 2 \leqslant \omega_2(B'') + 2 = \omega_2(B') + 1$$

and $\omega_2(B) = \omega_2(B'' - \{b\}) + 2 \geqslant \omega_2(B'') + 1 = \omega_2(B')$.

Hence, once again, we have $\omega_2(B') \leqslant \omega_2(B) \leqslant \omega_2(B') + 1$.

All possible cases have now been considered and the proof of the lemma by induction is complete.

We can now prove

Theorem 2

$\omega_3 \leqslant \omega_2$.

Proof

We proceed by induction on $|A|$. If $|A| = 0$ the result is clearly true. We assume then that $|A| > 0$ and the result holds for any set, B, s.t. $|B| < |A|$.

From the definition of selected set, $s_2(A) \subseteq s_3(A)$.

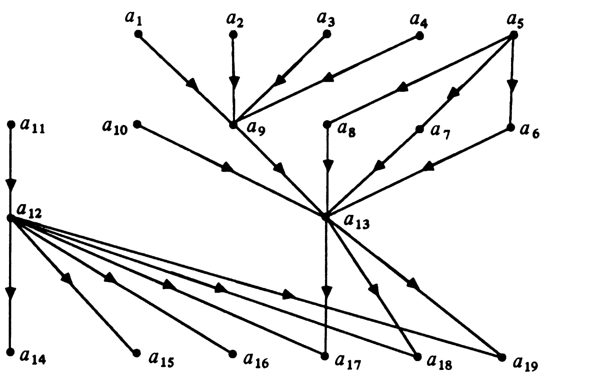
Case 1. $s_3(A) = s_2(A)$. Let $B = A - s_2(A)$. Then, by the induction hypothesis, $\omega_3(B) \leqslant \omega_2(B)$, but $\omega_3(A) = \omega_3(B)+1$ and $\omega_2(A) = \omega_2(B)+1$. Hence $\omega_3(A) \leqslant \omega_2(A)$.

Case 2. $s_3(A) = s_2(A) \cup \{a\}$ for some $a \in A$. Let $B = A - s_2(A)$. Then $\omega_3(A) = \omega_3(B-\{a\})+1 \leqslant \omega_2(B-\{a\})+1$ by the induction hypothesis. However, $\omega_2(B-\{a\}) \leqslant \omega_2(B)$ by Lemma 2 and $\omega_2(A) = \omega_2(B)+1$ and so $\omega_3(A) \leqslant \omega_2(A)$.

Hence the proof by induction is established.

We know from Theorem 1 that for arbitrary m

$$\omega_{m+1} \leqslant \left(2 - \frac{2}{m+1}\right)\omega_m \qquad (12)$$



(*a*) The dag.

	1	2	3	4	5
P_1	a_1	a_5	a_6	a_{13}	a_{17}
P_2	a_2	a_9	a_7	a_{14}	a_{18}
P_3	a_3	a_{10}	a_8	a_{15}	a_{19}
P_4	a_4	a_{11}	a_{12}	a_{16}	

(*b*) Gantt diagram for four processors.

	1	2	3	4	5	6
P_1	a_1	a_6	a_{11}	a_{12}	a_{14}	a_{19}
P_2	a_2	a_7	a_{13}	▨	a_{15}	▨
P_3	a_3	a_8	▨	▨	a_{16}	▨
P_4	a_4	a_9	▨	▨	a_{17}	▨
P_5	a_5	a_{10}	▨	▨	a_{18}	▨

(*c*) Gantt diagram for five processors.

Figure 3. Anomalous behaviour of level scheduling.

Theorem 2 shows that this bound is not the best possible for $m = 2$. However, it does show that for larger numbers of processors a move from m to $m+1$ processors can never increase processing time by more than a factor somewhat less than two. For large m, we can indeed get an anomalous slowdown approaching the value of the bound.

Having seen how badly an arbitrary list schedule can behave when increasing the number of processors, our next task is to consider the anomalous behaviour for the particular list-scheduling algorithms mentioned above. Two further list-scheduling algorithms are discussed in the following section.

In Fig. 3 we give an example to show anomalous behaviour with level scheduling. From this example it is easy to construct an example which demonstrates anomalous behaviour of the Coffman–Graham algorithm.

3. SCHEDULING AND PARALLEL COMPUTING

In applications where we are scheduling activities dynamically created (as in the case of functional programming), the whole dag is not available in advance. We could possibly estimate the eventual level of an activity and then apply Hu's algorithm, but the Coffman–Graham algorithm would be far too sophisticated for consideration. In practice, activities are simply scheduled as they are created. We consider now the two principal ways in which this can be achieved, breadth first (bf) and depth first (df), both generalisations of familiar tree-searching algorithms.[21]

We begin by defining the *depth* of an action in a dag. All *start actions*, i.e. actions without parents, are said to be at depth 0; the depth of any other action in the dag is then the length of the longest path to itself from a start action. For any total order, $\sqsubset$, on the set of actions A, in a dag, we define the *eldest parent*, r, of $p \in A$ by

$$r \in P = \{q \mid q \text{ is a parent of } p\}$$

and $$q \sqsubset r \;\forall q \in P, \quad q \neq r.$$

We then define the following two total orderings on a partially ordered set, A. If $p, q \in A$ then

$p\,bf\,q$ iff

(1) depth $(p) <$ depth (q) or

(2) depth $(p) =$ depth $(q) \neq 0$
and (a) eldest parent (p) *bf* eldest parent (q)
or (b) eldest parent $(p) =$ eldest parent (q) and p 'to the left of' q

or

(3) depth $(p) =$ depth $(q) = 0$ and p 'to the left of' q

and

$p\,df\,q$ iff

(1) depth $(p) >$ depth (q) or

(2) depth $(p) =$ depth $(q) \neq 0$
and (a) eldest parent (p) *df* eldest parent (q)
or (b) eldest parent $(p) =$ eldest parent (q) and p 'to the left of' q

or

(3) depth $(p) =$ depth $(q) = 0$ and p 'to the left of' q.

The relation 'to the left' is an arbitrarily chosen, but consistently applied rule. Given a diagrammatic representation of the poset, A, we can choose to interpret 'to the left of' in the obvious way. For example, the breadth-

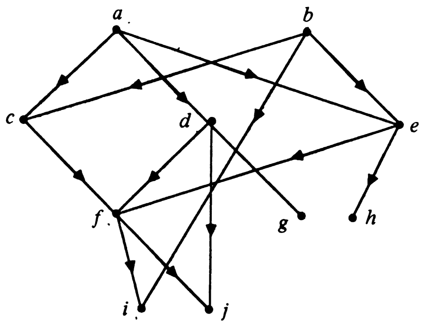


Figure 4.

first list for the dag in Fig. 4 is $a, b, d, c, e, g, f, h, i, j$ and the depth-first list is $i, j, g, f, h, d, c, e, a, b$.

From (9), we have a worst-case bound for breadth-first scheduling

$$\omega_{bf} \leqslant \left(2-\frac{1}{m}\right)\omega_0 \quad \text{for UET systems.} \qquad (13)$$

This bound is indeed achievable. Consider (m^2-m+1) start actions, the rightmost of which heads a chain of a total of m actions. With m identical processors and breadth-first scheduling, the first $m-1$ time units are spent on the leftmost m^2-m start actions. The next m time units are spent on the chain of tasks. The best strategy is to always work on the chain and then use the remaining $m-1$ processors to work on the independent activities; this gives an optimal schedule of length m.

Similarly, for depth-first scheduling we have a worst-case bound

$$\omega_{df} \leqslant \left(2-\frac{1}{m}\right)\omega_0 \quad \text{for UET systems} \qquad (14)$$

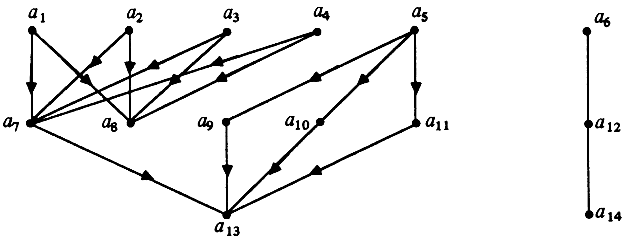
and, from the same example as we used for breadth-first, we see that this bound is also achievable. However, even though depth-first scheduling can perform as badly as any conceivable scheduling, it is the one that is usually used in the parallel implementation of functional programs. Depth-first has the additional drawback that it can exhibit anomalous behaviour as the number of processors is increased. Breadth-first does not have this anomalous behaviour, but has storage overheads that make its use impractical. An example demonstrating the anomalous behaviour of depth-first scheduling is given in Fig. 5. The depth-first list for this example is a_{13}, a_{14}, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_1, a_2, a_3, a_4, a_5, a_6. From a worst-case analysis, depth-first does not seem to do that much worse than a level strategy. However, on average, one would expect a level strategy to perform significantly better. This is a topic for further research, but the claim is certainly supported by all the evidence cited in Golovkin.[9]

We now demonstrate that breadth-first list scheduling guarantees that an increase in the number of processors will never increase the execution time.

Given a set of UET actions subject to a partial order $<$, let $a_1, a_2, \ldots, a_n$ be the breadth-first list of the actions in $(A, <)$ and let $E_{m,k}$ denote the set of actions that have been executed after k time units under breadth-first scheduling on m machines.

Lemma 3

$E_{m,k} = \{a_1, a_2, \ldots, a_{i_{m,k}-1}, a_{i_{m,k}}\}$ where $i_{m,k}$ is an integer between 1 and n.



(*a*) The dag.

P_1	a_1	a_7	a_9	a_{13}
P_2	a_2	a_8	a_{10}	a_{14}
P_3	a_3	a_5	a_{11}	▨
P_4	a_4	a_6	a_{12}	▨

(*b*) Gantt diagram for four processors.

P_1	a_1	a_7	a_{13}	a_{12}	a_{14}
P_2	a_2	a_8	a_6	▨	▨
P_3	a_3	a_9	▨	▨	▨
P_4	a_4	a_{10}	▨	▨	▨
P_5	a_5	a_{11}	▨	▨	▨

(*c*) Gantt diagram for five processors.

Figure 5.

Proof

The proof is by induction on k. Let $S = \{a_1, a_2, \ldots, a_s\}$ denote the set of start actions. In the first time unit, the m processors execute the actions $a_1, a_2, \ldots, a_{i_{m,1}}$ where $i_{m,1} = \min\{m, s\}$, so the lemma is true for $k = 1$.

Assume the lemma is true for $k \leqslant l$ and consider $k = l+1$. Let $R_{m,l}$ denote the readyset after l time units and let $a_{l_1}, a_{l_2}, \ldots, a_{l_r}$ be the breadth-first ordering of these actions, where $r = |R_{m,l}|$. If $l_j = i_{m,l}+j, l \leqslant j \leqslant r$, then the lemma is clearly true for $k = l+1$, with $i_{m,l+1} = i_{m,l} + \min\{m, r\}$, so assume j is the smallest integer such that

$$l_j \neq i_{m,l}+j \quad (j \in \{1, \ldots, r\}).$$

Since $1, 2, \ldots, i_{m,l}, i_{m,l}+1, \ldots, i_{m,l}+j-1$ are successive integers, it follows that

$$l_j > i_{m,l}+j.$$

Since $a_{i_{m,l}+j} \notin R_{m,l}$, eldest parent $(a_{i_{m,l}+j}) \notin E_{m,l}$. On the other hand, $a_{l_j} \in R_{m,l}$ implies eldest parent $(a_{l_j}) \in E_{m,l}$. But, by the inductive hypothesis, $E_{m,l}$ consists of the first $i_{m,l}$ actions in the breadth-first list of actions. Hence, eldest parent (a_{l_j}) *bf* eldest parent $(a_{i_{m,l}+j})$, and this implies a_{l_j} *bf* $a_{i_{m,l}+j}$, which contradicts the assumption that $a_1, a_2, \ldots, a_n$ is the breadth-first list.

The lemma is therefore true for $k = l+1$ and the proof by induction is complete.

Lemma 4

$i_{m,k} \leqslant i_{m+1,k}$.

Proof

The proof is again by induction on k. Since $i_{m,1} = \min\{m, s\}$, where s is the number of start actions, the lemma is true for $k = 1$. Assume that it is true for $k \leqslant l$ and consider $k = l+1$. As above, let $a_{l_1}, a_{l_2}, \ldots, a_{l_r}$ be the breadth-first ordering of the actions in the readyset $R_{m,l}$ and similarly let $a_{l'_1}, a_{l'_2}, \ldots, a_{l'_r}$ be the breadth-first ordering of the actions in the readyset $R_{m+1,l}$.

Now $i_{m,l+1} = i_{m,l} + \min\{m, r\}$ and $i_{m+1,l+1} = i_{m+1,l} + \min\{m+1, r'\}$. If $\min\{m+1, r'\} \geqslant \min\{m, r\}$ then it follows immediately from the inductive hypothesis that the lemma is true for $k = l+1$.

Consider $\min\{m, r\} > \min\{m+1, r'\}$. By the inductive hypothesis together with Lemma 3, it follows that

$$a_{l_i} \in E_{m+1,l} \cup R_{m+1,l}, \quad 1 \leqslant i \leqslant r.$$

Thus any action which will be executed on the m processor system during the next time unit has either already been executed on the $m+1$ processor system or will also be executed on that system during the next time unit. Hence,

$$i_{m+1,l+1} \geqslant i_{m,l+1}$$

and the proof by induction is complete.

We have therefore established the following theorem.

Theorem 3

If ω_m denotes the length of the breadth-first list schedule of $(A, <)$ on m processors then $\omega_{m+1} \leqslant \omega_m$.

4. CONCLUSION

We have shown that any list-scheduling algorithm will be within a small factor of the optimal pre-emptive scheduling algorithm. Since this is a worst-case performance, it encourages us to believe that any list-scheduling algorithm will be acceptable in a highly parallel system where incomplete information which can result from dynamic task creation prevents some of the common well-behaved scheduling algorithms from being used.

In addition, we have studied the problem of ensuring that more processors will result in faster processing. We have shown that with UET actions (which correspond to pre-emptive scheduling and integer execution times for actions) three processors will never be worse than two, provided the same list-scheduling algorithm is used. With breadth-first scheduling, this result can be strengthened so that $m+1$ processors are never worse than m, but this cannot be guaranteed for depth-first scheduling or for any other list-scheduling algorithm that we have considered.

Further work includes the scheduling of parallel processors under the more realistic assumption of interprocessor communication delays.

REFERENCES

1. F. W. Burton, Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Trans. on Prog. Lang. and Systems* **6**, 159–174 (1984).
2. F. W. Burton, Functional programming for concurrent and distributed computing, submitted for publication (1986).
3. N.-F. Chen, *An Analysis of Scheduling Algorithms in Multi-processing Computing Systems*. Technical Report UIUCDCS-R-75-724, Department of Computer Science, University of Illinois at Urbana, Champaign (1975).
4. N.-F. Chen and C. L. Liu, On a class of scheduling algorithms for multiprocessor computing systems. In *Parallel Processing* (Lecture Notes in Computer Science 24), edited T.-Y. Feng, pp. 1 – 16. Springer, Berlin (1975).
5. E. G. Coffman, Jr (ed.), *Computer and Job-Shop Scheduling Theory*. Wiley, New York (1976).
6. E. G. Coffman, Jr and R. L. Graham, Optimal scheduling for two processor systems. *Acta Informatica* **1**, 100–213 (1972).
7. M. A. H. Dempster, J. K. Lenstra and A. H. G. Rinooy Kan, *Deterministic and Stochastic Scheduling*. Reidel, Dordrecht, Holland (1982).
8. M. Fujii, T. Kasami and K. Nimomiya, Optimal sequencing of two equivalent processors. *SIAM Journal on Applied Mathematics* **17**, 784–789 (1969); erratum **20**, 141 (1971).
9. B. A. Golovkin, A comparison of methods of scheduling parallel computations in multiprocessor systems. *Engineering Cybernetics* **20**, 116–126 (1982).
10. R. L. Graham, Bounds on certain multiprocessing anomalies. *Bell System Technical Journal* **45**, 1563–1581 (1966).
11. R. L. Graham, Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics* **17**, 263–269 (1969).
12. R. L. Graham, Bounds on the performance of scheduling algorithms. In E. G. Coffman, Jr (Ref. 5).
13. E. C. Horvath, S. Lam and R. Sethi, A level algorithm for preemptive scheduling. *ACM Journal* **23**, 317–327 (1977).
14. T. C. Hu, Parallel sequencing and assembly line problems. *Operational Research* **9**, 841–848 (1961).
15. R. M. Karp, Reducibility among combinatorial problems. In *Complexity of Computer Computations*, edited R. E. Miller and J. W. Thatcher, pp. 85–103. Plenum Press, New York (1972).
16. T.-H. Lai and S. Sahni, Anomalies in parallel branch-and-bound algorithms. *Comm. ACM* **27**, 594–602 (1984).
17. S. Lam and R. Sethi, Worst case analysis of two scheduling algorithms. *SIAM Journal on Computing* **6**, 518–536 (1977).
18. C. L. Liu, Optimal scheduling on multi-processor computing systems. *Proceedings of the 13th Annual IEEE Symposium on Switching and Automata Theory*, pp. 155–160 (1972).
19. R. R. Muntz and E. G. Coffman, Jr, Optimal preemptive scheduling on two-processor systems. *IEEE Transactions, Computers* **C-18**, 1014–1020 (1969).
20. R. R. Muntz and E. G. Coffman, Jr, Preemptive scheduling of real time tasks on multiprocessor systems. *ACM Journal* **17**, 324–338 (1970).
21. N. J. Nilsson, *Problem-solving Methods in Artificial Intelligence*. McGraw-Hill, New York (1971).
22. R. Sethi, Algorithms for minimal-length schedules. In E. G. Coffman, Jr (Ref. 5).
23. J. D. Ullman, NP-complete scheduling problems. *Journal of Computer System Science* **10**, 384–393 (1975).
24. J. D. Ullman, Complexity of sequencing problems. In E. G. Coffman, Jr (Ref. 5).