1. Introduction

A number of problems in image processing, scientific computing, and numerical optimization are well suited for solution using parallel or pipelined computations. Typically there are far fewer processors than elements to be processed, and so some natural aggregation of elements is required for partitioning the workload among the processors. For example, a spatial aggregation of elements might be appropriate for image processing, whereas an aggregation based on contiguous columns might be appropriate for various matrix computations. The focus of the present paper is on this latter type of aggregation, in which a linear ordering of elements is imposed and an optimal partitioning of the elements is required that achieves a balanced workload among the processors.

One specific motivating example arises in studying parallel implementations of the Cholesky decomposition of a banded positive definite matrix A, as proposed by O'Hallaron (1988). The standard Cholesky decomposition $A = LL^T$ can be computed in a serial manner using two alternating stages. The first stage carries out an appropriate normalization of the current column; this information is then passed on to other columns during the second stage. In O'Hallaron's pipelined implementation, a set of contiguous columns (defined by the matrix bandwidth) is allocated to a smaller number of processors. Again, the Cholesky scheme consists of alternating stages of normalization and updating sets of columns. Because of the need for global synchronization at the end of each stage, the computational speed is limited by the processor having the maximum workload. The overall speed is then determined by the <u>sum</u> of the <u>maximum</u> workloads at each stage, and an allocation of columns to processors that minimizes this sum is desired.

Another specific motivating example occurs in the parallel solution of one-dimensional fluid flow problems using irregular grid hierarchies (see Berger and Oliger 1984). Nonuniform gridding is used to concentrate computational effort in domain regions where the solution changes rapidly (as from a shock or turbulence). In this grid hierarchy various uniformly spaced fine grids which do not span the entire domain are superimposed onto a uniformly spaced coarse grid which does. The "stages" of such problems correspond to the numerical integration of grids with a common spatial separation; inter-stage synchronization is used to enforce data dependencies between solutions at differing grid levels. The entire computation is parallelized by dividing the domain (or equivalently, the coarse grid points) among the given processors. The linear ordering of these grid points reflects the natural constraint that grid point updates depend on contiguous grid values. As in the Cholesky decomposition, the processor having the heaviest workload at a given stage limits the progress of the entire system. The sum of the maximum workloads at each stage again measures the computation's execution speed.

The following section provides a mathematical formulation of the general problem of minimizing an overall objective based on the sum of maximum workloads. This problem is termed the *Multistage Linear Array Assignment* (MLAA) problem. It is shown in Section 5 that the associated decision problem is NP-complete for a general number (r) of stages. Thus, we first concentrate on providing efficient polynomial algorithms for the MLAA when $r = 2$. Two separate approaches are developed (in Sections 3 and 4) for the two-stage problem. Section 6 discusses how one of these algorithms can be generalized in a natural way to r stages, and it also develops heuristic procedures for an arbitrary number of stages. Computational experience with the exact and heuristic algorithms is presented in Section 7.

2. Formulation

It is supposed that n elements or *modules* are given, each of which has two associated nonnegative processing times $x_i, y_i \geq 0$ $(i = 1, \ldots, n)$. There are also available p processors, $p < n$, each of which is capable of handling an arbitrary number of modules. We consider a partition of the set of modules $\{1, 2, \ldots, n\}$ into p intervals $I_1, I_2, \ldots, I_p$ where each interval consists of consecutive modules $\{t, t+1, \ldots, t+k\}$, $k \geq 0$. A (quite reasonable) assumption made here is that the processing time for each interval is additive in the processing times of its component modules:

$$x(I_j) = \sum_{k \in I_j} x_k , \quad y(I_j) = \sum_{k \in I_j} y_k .$$

The objective here is to select intervals $I_1, \ldots, I_p$ for the p processors to minimize $z(I_1, \ldots, I_p)$, where

$$z(I_1, \ldots, I_p) = [\max_j x(I_j)] + [\max_j y(I_j)] . \qquad (1)$$

The extension to r processing times (stages) is immediate.

This problem can be formulated on an acyclic network $G = (N, E)$, where the node set $N = \{0, 1, \ldots, n\}$ and the edge set E consists of all edges (i, j) with $i < j$. Each edge (i, j) has the associated *weights* x(i, j) and y(i, j), defined by

$$x(i, j) = \sum_{i < k \leq j} x_k , \quad y(i, j) = \sum_{i < k \leq j} y_k . \qquad (2)$$

Notice that G has $m = O(n^2)$ edges.

It is easily seen that our two-stage problem is precisely that of finding in G a path of exactly p edges from node 0 to node n that minimizes the sum of the maximum edge weight in each component. This "min-sum-max" problem is a bicriterion path problem that has not been extensively studied, although related bicriterion path problems have been investigated. Warburton (1985) considers the "min-max-sum" problem and demonstrates that it is NP-hard for $r = 2$ components. The problem is strongly NP-hard for general r. A number of authors (Hansen 1980, Henig 1985) have studied the problem of generating "efficient" (Pareto optimal) paths in bicriterion networks, using a variety of path length measures. The most pertinent measure of path length to our present study is the maximum edge weight for each criterion. A path is termed *efficient* when no other admissible path has a smaller value for one criterion without a larger value for the other. Polynomial-time algorithms for this "minmax-minmax" bicriterion path problem have been given by Hansen (1980), Berman et al. (1987), and Warburton (1987). Other related work is presented by Bokhari (1988), relative to the single-objective problem of minimizing the maximum of the maximum x-weight and the additive y-weight, in a doubly weighted network. While the problem studied here is significantly different from these other problems, certain common solution strategies exist that have been exploited in our algorithmic development of the next sections.

Before discussing algorithms for the two-stage case, it is worthwhile to point out certain difficulties intrinsic to the problem of minimizing the objective function (1). Consider the two-stage problem with processing times x_i, y_i given in Table 1, with $p = 3$. The (unique) optimal partition for this problem has $I_1 = \{1\}$, $I_2 = \{2\}$, $I_3 = \{3, 4\}$. However, in the subproblem defined over the modules $I_2 \cup I_3 = \{2, 3, 4\}$, the optimal partition for $p = 2$ processors uses $I_2 = \{2, 3\}$ and $I_3 = \{4\}$. Thus the principle of optimality required for a dynamic programming formulation does not hold. In other words, optimal solutions cannot be found simply by extending (in the best possible way) the known optimal solutions for smaller subproblems.

Table 1

A Three Processor Example that Violates the Optimality Principle

	Module (i)			
	1	2	3	4
x_i	8	6	6	4
y_i	11	3	5	6

3. The Labeling Algorithm

In this section a labeling algorithm is developed to solve the two-stage MLAA problem (1). This algorithm works directly on the network G introduced in the previous section and exploits the fact that G is acyclic. Since the principle of optimality cannot be invoked for this problem to extend optimal paths for a subproblem, we maintain at each node $j \in N$ several sets of candidate paths. That is, associated with each node j will be certain sets L(j; k) of vectors in $\mathbf{R}^2$. The set

$L(j;k) = \{(x^1, y^1), (x^2, y^2), \ldots, (x^t, y^t)\}$ corresponds to "path length vectors" for paths P from node 0 to node j having exactly k edges. In this context, the first component x^s of such a path vector is simply the maximum x-weight of path P, and the second component y^s is the maximum y-weight of path P.

To express these notions more precisely, let $u, v \in \mathbf{R}^2$ be vectors with $u = (u_1, u_2)$ and $v = (v_1, v_2)$, and define the product $u \otimes v$ by

$$u \otimes v = (\max(u_1, v_1), \max(u_2, v_2)) \ .$$

In this notation, the path length vector for path P is given by

$$\mathrm{len}(P) = \otimes\prod_{(i,j)\in P} (x(i,j), y(i,j))\ ,$$

and the label L(j; k) on node j is given by

$$L(j;k) = \{\mathrm{len}(P) \text{: P is a path from 0 to j with } \|P\| = k\}. \qquad (3)$$

Here x(i, j) and y(i, j) are as defined in (2), and $\|P\|$ is used to denote the *cardinality* (number of edges) of path P. It will also be convenient to extend the definition of $\otimes$ to sets of vectors: if $S \subseteq \mathbf{R}^2$ and $v \in \mathbf{R}^2$ then

$$S \otimes v = \{w \otimes v \colon w \in S\}.$$

Throughout, we will maintain sets of path vectors for each node. At the end of the process, an optimal path vector to node n will be identified, from which it is straightforward to recover an associated path (via a simple backtracking scheme). Consequently, we shall work entirely with path vectors. Not all possible path vectors need to be retained at a node, however. To this end, a vector x is said to *dominate* vector y if $x \le y$ holds componentwise and at least one of the inequalities is strict. (This nomenclature is meant to convey the idea that x is "better" than y; however usage in this way is not standard in the literature.) A set S of vectors is *efficient* (Pareto optimal) if no vector $x \in S$ dominates another vector $y \in S$. More generally, the set EFFICIENT(S) is obtained from set S by removing all dominated vectors.

In the algorithm given below, only efficient sets $L(j;k)$ are maintained at each node. This algorithm represents a suitably modified version of the algorithm ACYCLIC given by Warburton (1987) for obtaining the set of efficient path vectors in a bicriterion shortest path problem.

LABELING ALGORITHM

```
1.  $L(0;0) = \{(0, 0)\}$;
    $L(0;k) = \emptyset$, for $k = 1, \ldots, p$.

2.  For $j = 1, \ldots, n-1$
      $L(j;0) = \emptyset$;
      For $k = 1, \ldots, p-1$
          $L(j;k) = \text{EFFICIENT}\{\bigcup_{i<j} L(i;k-1) \otimes (x(i, j), y(i, j))\}$.

3.  $L(n;p) = \text{EFFICIENT}\{\bigcup_{i<n} L(i;p-1) \otimes (x(i, n), y(i, n))\}$.

4.  Select $v = (v_1, v_2) \in L(n;p)$ for which $v_1 + v_2$ is minimum.
```

Upon termination, the labeling algorithm produces a set of efficient path vectors $L(n;p)$, from which an optimal p-edge path vector for (1) is readily obtained in Step 4. If we also maintain the vector labels $L(n;k)$ at node n, sensitivity analysis information will be available on the effect of reducing the number of processors from the fixed value p. Of course, optimal partitions for the various subproblems using modules $\{1, \ldots, j\}$, $j < n$, are also readily derived from the vector labels on node j. It should be noted that the range for k in Step 2 can actually be replaced by $k = \max\{1, p-n+j\}, \ldots, \min\{p-1, j\}$. This type of pruning allows the elimination of paths whose cardinality is too small to lead to an optimal path at node n having p edges, and it also avoids the examination of lists known to be empty.

The worst-case time complexity of the labeling algorithm is determined by the complexity of calculating EFFICIENT(S) in Steps 2 and 3. Suppose that L_{max} is the maximum size of any set

$L(j;k)$ produced during the course of the algorithm. By keeping each $L(j;k)$ as a list of vectors, ordered by their first component, repeated merge sorts suffice to maintain an ordered list without dominated elements. In the worst case there can be $O(n)$ sublists to merge, each of size L_{max}, resulting in $O(n \log n\ L_{max})$ work to calculate EFFICIENT(S). Overall, the complexity of carrying out Steps 2 and 3 is then $O(p\ n^2 \log n\ L_{max})$. Since each element of the efficient set $L(j;k)$ represents a path vector len(P) and since the first component of len(P) can assume at most $m = |E|$ distinct values, $L_{max} \leq m = O(n^2)$ and the overall time complexity of the labeling algorithm can be bounded by $O(p\ n^4 \log n)$. Space requirements are $O(p\ n\ L_{max}) \leq O(p\ n^3)$ in the worst case. In practice, however, these estimates would appear to be overly pessimistic and the expected performance of the algorithm should be significantly better. The empirical results presented in Section 7 confirm this belief.

4. The Recursive Bisection Algorithm

This section describes a different approach for solving the two-stage problem, based on generating a set of candidate path vectors guaranteed to contain all efficient path vectors from node 0 to node n. The optimal solution vector can then be easily obtained from this set. The following simple observation serves to motivate this second approach. Namely, if P^* is an optimal path for the two-stage problem, then the optimal objective function value is given by

$$z^* = [\max_{a \in P^*} x(a)] + [\max_{b \in P^*} y(b)] \equiv x(a^*) + y(b^*) ,$$

where a^* and b^* are edges of P^*. Consider the subgraph of G

$$G_a = \{e \in E: x(e) \leq x(a)\}$$

consisting of all edges with x-weight at most x(a). Then it is not difficult to find a "best" path in G_a with respect to the y-weights, a path P_a in G_a from 0 to n with cardinality p that minimizes the maximum y-weight along P_a:

$$y(b_a^*) = \max_{b \in P_a} \{y(b)\} .$$

Such a minimax path can be found using a straightforward modification of the standard Dijkstra shortest path algorithm (see Kalaba 1964). Finally, the optimal two-stage solution is found by selecting a path P_a that minimizes $x(a) + y(b_a^*)$. This basic approach would require m calls to the minimax (p-edge) path routine, each such invocation requiring $O(p\ m)$ work, so the overall time complexity is bounded by $O(p\ m^2) = O(p\ n^4)$.

An improved approach that treats the x-weights and y-weights more symmetrically, and that has a significantly better complexity bound, will now be developed. First, the x-weights on the m arcs of G are sorted into nondecreasing order: $x(a_1) \le x(a_2) \le \ldots \le x(a_m)$. Likewise, the y-weights are sorted as $y(b_1) \le y(b_2) \le \ldots \le y(b_m)$. Next, we define the subgraph $G_{a,b}$ of G via

$$G_{a,b} = \{e \in E: x(e) \le x(a),\ y(e) \le y(b)\}.$$

The MLAA problem can then be solved by finding edges $a, b \in E$ so that $[x(a), y(b)]$ is *feasible* (i.e., $G_{a,b}$ admits a path P from 0 to n with $\|P\| = p$) and $x(a) + y(b)$ is *minimum* (over all feasible pairs).

It will be useful to introduce a *feasibility matrix* Z, whose (i, j) entry z_{ij} indicates whether the pair $[x(a_i), y(b_j)]$ is feasible (F) or not (I). Because of the previous ordering of x-weights and y-weights, whenever $z_{st} = F$ then $z_{ij} = F$ for $i \ge s$ and $j \ge t$. Also, if $z_{st} = I$ then $z_{ij} = I$ for $i \le s$ and $j \le t$. A typical pattern for the matrix Z is illustrated in Figure 1. The efficient path vectors $[x(a), y(b)]$ are indicated by circled entries in the matrix.

$$Z = \begin{bmatrix} I & I & I & \text{\textcircled{F}} & F & F \\ I & I & I & F & F & F \\ I & I & \text{\textcircled{F}} & F & F & F \\ \text{\textcircled{F}} & F & F & F & F & F \\ F & F & F & F & F & F \\ F & F & F & F & F & F \end{bmatrix}$$

Figure 1. Example feasibility matrix Z.

We would like to generate a candidate set of entries (at most m), guaranteed to contain the efficient (circled) entries in the feasibility matrix. Since it is computationally expensive to determine the values of entries in this matrix, it is desirable to evaluate as few entries as possible. It is somewhat surprising that we can in fact determine the candidate set (having possibly m entries) by evaluating only at most O(m) of the entries z_{ij}.

For notational convenience, it will be assumed that $m = 2^k$ for some integer $k \geq 0$. Because of the ordering properties of Z, derived from the ordering of x-weights and y-weights, we can perform a binary search on the elements of the (median) row $s = \frac{m}{2}$ of Z to find the smallest t so that $z_{st} = F$. This will be termed a binary search at *level* 0, and it requires O(log m) evaluations of entries in Z. Entry (s, t) of the matrix then becomes a candidate solution. The ordering properties of Z ensure that the submatrix of Z with upper left corner at (s, t) contains only F's, while the submatrix with lower right corner at (s, t-1) contains only I's. Therefore, it is only necessary to explore two remaining submatrices for candidate solutions: the submatrix with lower left corner at (s-1, t), and the submatrix with upper right corner at (s+1, t-1). Each submatrix can in turn be explored by performing a binary search along a row in median position within the submatrix. If the first submatrix has c_1 columns and the second has c_2 columns, then the number of evaluations needed to carry out these two binary searches is $O(\log c_1) + O(\log c_2)$. By the concavity of the log function, $\frac{1}{2}\log c_1 + \frac{1}{2}\log c_2 \leq \log\left[\frac{1}{2}(c_1 + c_2)\right] = \log\frac{m}{2}$, and so the number of evaluations required for these two level 1 searches is of the order $2\log\frac{m}{2}$. In a similar way, the 2^l binary searches at level l require at most $2^l \log\frac{m}{2^l}$ evaluations, so the total number of evaluations needed is of the order

$$\log m + 2\left(\log\frac{m}{2}\right) + 4\left(\log\frac{m}{4}\right) + \ldots + \frac{m}{4}(2) + \left(\frac{m}{2}\right)1 \leq m\sum_{i=1}^{\infty}\frac{i}{2^i}.$$

Since the infinite series is convergent, at most O(m) entries of Z will need to be examined. A formal statement of this algorithm (RECURSIVE BISECTION) is given below, applied to an $m \times m$ feasibility matrix Z.

RECURSIVE BISECTION ALGORITHM

[Main Program]

```
    $C = \varnothing$;
    BISECT (Z, m, m, C);
    Select $(i, j) \in C$ with minimum $x(a_i) + y(b_j)$.
```

[Recursive Subprogram]

```
    procedure BISECT (M, rows, cols, C)

        $s = \lceil rows/2 \rceil$;
        Determine (by binary search) the smallest t
            so that $m_{st} = F$; $C = C \cup \{(s, t)\}$;
        if $s > 1$ then
            Let $M_1$ be the submatrix of M with lower left
                corner at (s-1, t);
            BISECT ($M_1$, s-1, cols-t+1, C);
        if $s <$ rows and $t > 1$ then
            Let $M_2$ be the submatrix of M with upper right
                corner at (s+1, t-1);
            BISECT ($M_2$, rows-s, t-1, C).
```

As shown above, this algorithm requires O(m) entries of Z to be evaluated in carrying out the various binary searches on partial rows of the input matrix. Once the smallest feasible entry in the prescribed row is determined, the candidate set C is updated appropriately. We now discuss how any entry of the feasibility matrix can be efficiently computed, when required.

Recall that the value of an entry in the feasibility matrix is determined by whether there is a path in $G_{a,b}$ of cardinality p from 0 to n. In view of the definition (2) of edge weights and the nonnegativity of processing times, the following monotonicity property is immediate.

Property 1. If $i \le j \le k$ then $x(i, j), x(j, k) \le x(i, k)$ and $y(i, j), y(j, k) \le y(i, k)$.

The next property follows directly from Property 1.

Property 2. Suppose that $i \le j \le k$ and that $(i, k) \in G_{a,b}$. Then $(i, j) \in G_{a,b}$ and $(j, k) \in G_{a,b}$.

Proof: Since $(i, k) \in G_{a,b}$ we have $x(i, k) \le x(a)$ and $y(i, k) \le y(b)$. From Property 1, $x(i, j) \le x(i, k) \le x(a)$ and $y(i, j) \le y(i, k) \le y(b)$ and so $(i, j) \in G_{a,b}$. In a similar way, $(j, k) \in G_{a,b}$.

We are now in a position to describe an algorithm for determining the feasibility of $[x(a), y(b)]$. The basic idea is to start from node 0 and to repeatedly follow the edge in $G_{a,b}$ that reaches to the node of largest index, until we reach node n. If this constructed path has cardinality $\le p$ then it will turn out that $[x(a), y(b)]$ is feasible, and conversely. This greedy procedure (REACH) for checking feasibility in $G_{a,b}$ is stated below.

```
procedure REACH(a, b)

1. s = 0; card = 0;
2. While card < p
       Find the largest j so that $(s, j) \in G_{a,b}$;
       card = card + 1;
       if j = n then output *true* and stop
           else s = j;
3. Output *false*.
```

The validity of REACH is now established: namely, it will be shown that the output *true* is produced precisely when there is a path of cardinality p in $G_{a,b}$ from node 0 to node n. Suppose that REACH outputs *true*. Then we are assured of a path P of cardinality $\le p$ in $G_{a,b}$ from 0 to n. Since $p < n$, Property 2 ensures that if such a path actually has cardinality $< p$, then there must be some path of cardinality exactly p. (Just successively replace edges $(i, j) \in P$ by edges (i, k) and (k, j) as needed.) On the other hand, suppose there is a path Q in $G_{a,b}$ of cardinality p from 0 to n, where $Q = [(i_0, i_1), (i_1, i_2), \ldots, (i_{p-1}, i_p)]$, with $i_0 = 0$ and $i_p = n$. Let the path produced by

REACH be $P = [(j_0, j_1), (j_1, j_2), \ldots, (j_{t-1}, j_t)]$, with $j_0 = 0$. Clearly, since $i_0 = j_0$ and $(i_0, i_1) \in G_{a,b}$ procedure REACH will generate $j_1 \geq i_1$. Since $(i_1, i_2) \in G_{a,b}$ Property 2 guarantees that $(j_1, i_2) \in G_{a,b}$ and thus the node j_2 selected by REACH must satisfy $j_2 \geq i_2$. Continuing in this way, we can establish that $j_3 \geq i_3, \ldots, j_p \geq i_p = n$. Hence REACH will detect a path of cardinality at most p to node n and output *true*.

The time complexity of REACH, and consequently that of the recursive bisection algorithm, is now easy to establish. First, we note that the graph $G_{a,b}$ need not be explicitly constructed in order to carry out Step 2 of REACH. Namely, by Property 1 the edges emanating from node s are automatically ordered by their x-weight and by their y-weight:

$$x(s, s+1) \leq x(s, s+2) \leq \ldots \leq x(s, n); \quad y(s, s+1) \leq y(s, s+2) \leq \ldots \leq y(s, n).$$

Thus, a binary search can be used on the $n - s$ pairs $(s, s+1), \ldots; (s, n)$ to find the largest index j for which $x(s, j) \leq x(a)$ and $y(s, j) \leq y(b)$. The time complexity of one iteration of Step 2 is thus $O(\log n)$, and REACH therefore can be implemented to run in $O(p \log n)$ time. As a result, the recursive bisection algorithm has $O(p\, m \log n) = O(p\, n^2 \log n)$ worst-case time complexity. Also, the initial sorting of x-weights and y-weights can be done in $O(m \log m) = O(n^2 \log n)$ time, so the overall time complexity remains at $O(p\, n^2 \log n)$. The space requirements are dominated by the $O(n^2)$ storage needed to keep G as well as the sorted x-weights and y-weights.

In summary, the recursive bisection method presented in this section exhibits a better worst-case time complexity than the labeling algorithm of Section 3. Because the former algorithm makes repeated use of binary searches (whose worst-case and average-case behaviors are similar), it is anticipated that in practice the empirical complexity of the algorithm should closely reflect its worst-case bound. This is in contrast to the labeling algorithm, whose empirical performance will depend on the number of labels (L_{max}) actually encountered.

A final observation concerns the additive assumption made for combining processing times into edge weights $x(i, j)$ and $y(i, j)$. The specific (additive) way of combining the x_k's and the y_k's in (2) is by no means essential. In fact, any monotone nondecreasing function of the processing times can be used. Then Properties 1 and 2 will continue to hold and the resulting algorithm will still be valid.

5. Complexity Results

In Section 2 we formulated the MLAA problem as a min-sum-max optimization problem on a graph. For the case when exactly two stages are associated with each module, two polynomial-time algorithms for this problem were presented in Sections 3 and 4. In the present section we prove that for r stages the MLAA decision problem is NP-complete. To do this we show that any *3-Satisfiability* decision problem, a well known NP-complete problem (Garey and Johnson 1979), can be reduced to an instance of the MLAA decision problem. Before proceeding with the proof we state the aforementioned problems formally.

3-SAT

Instance: A collection $C = \{c_1, c_2, \ldots, c_m\}$ of clauses on a finite set $U = \{x_1, x_2, \ldots, x_u\}$ of 0,1 variables such that each clause c_i is a disjunction of exactly three literals (x_i or $\overline{x}_i$ is a literal).

Decision Problem: Is there a truth assignment for U that satisfies all the clauses in C?

MLAA

Instance: A collection of n modules and p processors. Associated with each module i is an r dimensional nonnegative real weight vector $(w_1(i), w_2(i), \ldots, w_r(i))$.

Decision Problem: Is there a partition of the n modules into p contiguous intervals $I_1, \ldots, I_p$ so that $\sum_{k=1}^{r} \max_j w_k(I_j) \leq B$, where B is a positive real number? (As in Section 2, $w_k(I_j) = \sum_{i \in I_j} w_k(i)$.)

Theorem 1. The MLAA decision problem is NP-complete.

Proof: It is easy to see that MLAA is a member of class NP since a nondeterministic turing machine need only guess a partition $I_1, \ldots, I_p$ and check in polynomial time whether or not $\sum_{k=1}^{r} \max_j w_k(I_j) \leq B$.

It remains to show that any instance of 3-SAT can be transformed into a MLAA problem. For the purposes of the proof, we define $B = 30u$, $p = 3u + 4m$, $n = 4u + 6m$, $r = 2u + 1$, and $F = 24u$. Here F represents a parameter whose use will become apparent as the proof proceeds. Throughout the proof we will use the term *bottleneck weight* to refer to $\max_j w_k(I_j)$ relative to some entry k. The transformation consists of two parts. The first involves constructing four consecutive modules for each $x_i \in U$; the second appends a further 6m modules (described later) corresponding to the m clauses. Associate with each of the first 4u modules a weight vector of dimension r. The first 2u entries of each weight vector will be positionally associated with the elements of U as follows: $(x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_u, \overline{x}_u)$. Entry $2u + 1$ is not associated with a literal and will contain some fractional part of F. This entry is used to force particular assignments of modules to processors. In particular, the values chosen for F, B, and p will force a bottleneck weight of F in entry $2u + 1$ for any feasible partition of the modules.

Each set of four weight vectors associated with a variable x_i are assigned values in a similar fashion. Module $j \equiv 1 \pmod 4$ corresponds to the start of a new set of four modules associated with a variable. Assume, without loss of generality, that $j \equiv 1 \pmod 4$ corresponds to the ith variable. Each entry of the weight vector for module j will be 0, except for entry $2u + 1$ which has a value of F. The weight vector for module $j + 1$ has a 2 in entry $2i - 1$, an $F/2$ in entry $2u + 1$, and 0's elsewhere. The weight vector for module $j + 2$ has a 2 in entries $2i - 1$ and $2i$, an $F/2$ in entry $2u + 1$, and 0's elsewhere. Lastly, the weight vector for module $j + 3$ has a 2 in entry $2i$, an $F/2$ in entry $2u + 1$, and 0's elsewhere.

The values given to F, B and p insure that modules $j \equiv 1 \pmod 4$ are assigned to a single processor and modules $j + 1$, $j + 2$, and $j + 3$ are assigned to exactly two processors. In particular if modules $j + 1$ and $j + 2$ are assigned to the same processor then bottleneck weights of 4 and 2 result for the components associated with x_i and $\overline{x}_i$ respectively. We interpret this as indicating that x_i is *true* since 4 is greater than 2. The alternative assignment indicates $\overline{x}_i$ is *true*.

The second part of the transformation concerns clauses. We construct a sequence of 6 consecutive modules for each clause in an instance of 3-SAT. Again we associate a weight vector of length r with each of the 6m modules. The construction of these weight vectors follows a similar pattern for each of the m six-module sets. Let $v \equiv 1 \pmod 6$ and $j = 4u + v$. The weight vector for module j has 0's in entries 1 to 2u and an F in entry $2u + 1$. The weight vector for module $j + 1$ has a 2 for the entry corresponding to the first literal in clause $c = \lceil (v - 1)/6 \rceil + 1$, a $2F/3$ in entry $2u + 1$, and 0's elsewhere. The weight vector for module $j + 2$ has a 1 for the entries corresponding to the first two literals of clause c, an $F/3$ in entry $2u + 1$, and 0's elsewhere. The weight vector for module $j + 3$ has a 1 for the entry corresponding to the second literal of clause c, an $F/3$ in entry $2u + 1$, and 0's elsewhere. The weight vector for module $j + 4$ has a 1 for the entries corresponding to the last two literals of clause c, an $F/3$ in entry $2u + 1$, and 0's elsewhere. Lastly, the weight vector for module $j + 5$ has a 2 for the entry corresponding to the third literal of clause c, a $2F/3$ in entry $2u + 1$, and 0's elsewhere.

In view of the values chosen for F, B, and p, each module $j = 4u + v$, with $v \equiv 1 \pmod 6$ must be assigned to a single processor and modules $j + 1, \ldots, j + 5$ must be partitioned among exactly three processors. Any feasible partition of modules $j + 1, \ldots, j + 5$ into three intervals must have a bottleneck weight of no more than F in entry $2u + 1$. In addition, such a feasible partition will result in a bottleneck weight of 3 for at least one (possibly two) of the entries (literals). Any literal associated with a bottleneck weight of 3 is considered as one satisfying the clause. Note that if the literal was assigned the value *true* earlier in the variable to module assignment, that entry will already have a bottleneck weight of 4 which will mask the 3. However, if a literal selected by the clause was not assigned the value *true* earlier, that entry's bottleneck weight of 2 will be increased by the value 3 created in the modules associated with the clauses.

This transformation of an instance of 3-SAT into an instance of MLAA is best illustrated by an example. Suppose $S = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$ is the given 3-SAT problem, so that $U = \{x_1, x_2, x_3\}$, $u = 3$, $m = 2$, $n = 24$, $B = 90$, $F = 72$, and $p = 17$. The module weight vectors corresponding to the three elements of U are given in Figure 2 and the module weight vectors corresponding to the clauses are given in Figure 3. One solution for S has x_1, x_2, and $\bar{x}_3$ *true*. A module to processor assignment corresponding to this solution is given in Figure 4. The bottleneck vector associated with the solution in Figure 4 is (4, 2, 4, 2, 2, 4, F). The component-wise sum of this vector is $F + 18 = B$, as expected.

Finally we must show that a given instance of 3-SAT is solved if and only if the corresponding instance of MLAA has a solution with a cost of no more than $B = 30u$. Suppose that a given instance of 3-SAT has a solution. Our prior comments guarantee that the instance of MLAA constructed from the given instance of 3-SAT has an assignment for which the sum of the x_i and $\bar{x}_i$ bottleneck weights is 6 for all $i = 1, \ldots, u$ and the bottleneck weight in the last entry is F. Therefore we have a solution with total weight $6u + F = B$. No solution to the constructed instance of MLAA with cost less than B is possible since entry $2u + 1$ will always contribute a bottleneck weight of F and the bottleneck weights corresponding to x_i and $\bar{x}_i$ contribute a sum no less than 6, for a total weight of at least B. It follows then that any solution to the instance of MLAA will imply a variable assignment that is a solution for the corresponding instance of 3-SAT.

	1	2	3	4	5	6	7	8	9	10	11	12
x_1	0	2	2	0	0	0	0	0	0	0	0	0
$\bar{x}_1$	0	0	2	2	0	0	0	0	0	0	0	0
x_2	0	0	0	0	0	2	2	0	0	0	0	0
$\bar{x}_2$	0	0	0	0	0	0	2	2	0	0	0	0
x_3	0	0	0	0	0	0	0	0	0	2	2	0
$\bar{x}_3$	0	0	0	0	0	0	0	0	0	0	2	2
•	F	F/2	F/2	F/2	F	F/2	F/2	F/2	F	F/2	F/2	F/2

Figure 2. Module weights corresponding to the variables of U.

	13	14	15	16	17	18	19	20	21	22	23	24
x_1	0	2	1	0	0	0	0	0	0	0	0	0
$\bar{x}_1$	0	0	0	0	0	0	0	2	1	0	0	0
x_2	0	0	0	0	0	0	0	0	1	1	1	0
$\bar{x}_2$	0	0	1	1	1	0	0	0	0	0	0	0
x_3	0	0	0	0	1	2	0	0	0	0	1	2
$\bar{x}_3$	0	0	0	0	0	0	0	0	0	0	0	0
•	F	2F/3	F/3	F/3	F/3	2F/3	F	2F/3	F/3	F/3	F/3	2F/3

Figure 3. Module weights corresponding to the clauses.

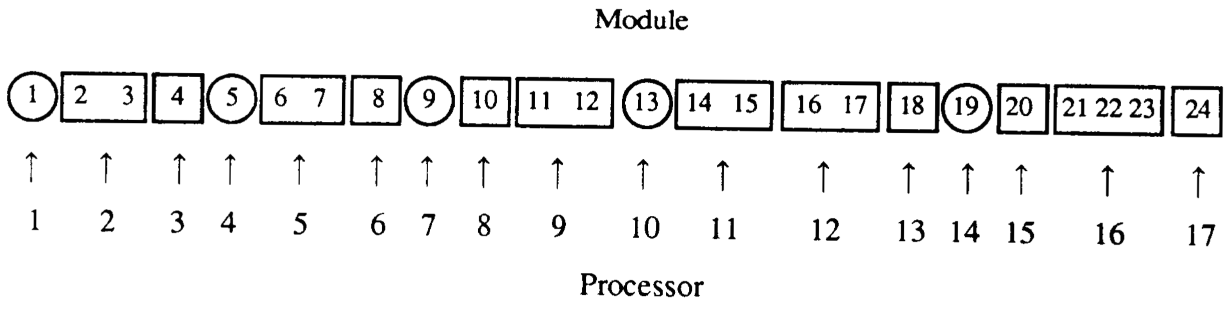

Figure 4. A solution to S and the module to processor assignment.

Since we can solve the MLAA decision problem by finding a min-sum-max path with p edges in an acyclic graph, it follows that the min-sum-max path decision problem is NP-complete for acyclic graphs. We state this as a corollary.

Corollary 1. The Min-Sum-Max decision problem for acyclic graphs is NP-complete.

The NP-completeness of the MLAA decision problem depends on treating r as a problem variable. For fixed r the natural extension of the labeling algorithm in Section 3 has polynomial-time complexity. To describe this extension, the only change needed from our previous description is in the efficient set calculation. The following discussion centers on the labeling algorithm pseudocode and complexity analysis given in Section 3.

Let $W(i, j) = (w_1(i, j), w_2(i, j), \ldots, w_r(i, j))$ be the weight vector associated with the edge between nodes i and j. Each computation of the form $L(i;k-1) \otimes W(i, j)$ performed in Steps 2 and 3 requires $O(r\, L_{max})$ time; constructing the list $L_j = \bigcup_{i<j} L(i;k-1) \otimes W(i, j)$ requires $O(n\, r\, L_{max})$ time. Notice that L_j has length $\bar{L} = O(n\, L_{max})$. Kung et al. (1975) describe an $O(\bar{L} \log^{r-2} \bar{L})$ time algorithm (for $r > 2$) that can be used to calculate EFFICIENT(L_j). Since L_{max} has $O(m^{r-1})$ length, a fact easily shown by induction, this extended labeling algorithm has $O(p\, n^{2r} \log^{r-2} (n^{2r-1}))$ time complexity. For fixed r, MLAA can thus be solved in polynomial time, albeit a polynomial of high degree. The following section outlines certain refinements for improving the performance of this type of algorithm.

6. Algorithms for r > 2

The recursive bisection algorithm of Section 4 can also be generalized for $r > 2$ stages: REACH extends in a natural way, and the feasibility matrix becomes an r-dimensional array. However, the search process does not scale well. In the two-stage problem, knowledge that (x, y) is the least costly feasible solution in a row allows us to ignore at least one-half of the rectangle being searched and creates two smaller rectangles to search recursively. In an r-stage problem we may similarly find the least costly feasible solution within a "row" (vectors that differ only in a given dimension). However, this identification allows us to eliminate as little as $1/2^{r-1}$ of the hyper-rectangle being searched and creates up to 2^r-2 hyper-rectangles that must be searched recursively. Consequently the remainder of this section focuses on extensions of the labeling algorithm for $r > 2$. Specifically, we describe computational refinements of the labeling algorithm, together with two heuristics suitable for problems that are too large for exact solution.

We have tested two algorithms for determining EFFICIENT(L_j): the algorithm of Kung et al. (1975) and an algorithm that does domination checking on the fly. On all the problems tested, the former algorithm ran an order of magnitude slower. Therefore, the less sophisticated (but for our problems, apparently faster) second approach for calculating $L(j;k)$ has been subsequently used. This procedure constructs, then accepts or rejects, each potential member of $L(j;k)$. An accepted member may cause previously accepted members to be discarded, and it is entered only once into $L(j;k)$. A potential member is immediately rejected if it is found to be dominated by an existing member of $L(j;k)$. This algorithm, listed below, requires $O(n\, r\, L_{max}^2)$ time to construct $L(j;k)$.

CONSTRUCTION ALGORITHM

```
1.  $L(j;k) = \varnothing$;
2.  For  $i = 1, \ldots, j-1$
3.      For all $U \in L(i;k-1)$
            $T = U \otimes W(i,j)$;
            For all $V \in L(j;k)$
                if $T \leq V$
                    $L(j;k) = L(j;k) - \{V\}$
                else if $T \geq V$
                    go to 3;
            $L(j;k) = L(j;k) \cup \{T\}$.
```

Let $T \in L(j;k)$ and suppose there is a path vector $X \in L(j;h)$, $h < k$, such that $X \leq T$ (componentwise vector comparison). If P and Q are subpaths corresponding to path vectors T and X respectively, then in any solution path (with p edges) that uses P, the subpath P can be replaced with subpath Q, yielding a new solution path (with < p edges) having the same or an improved solution cost. It follows from Property 1 of Section 4 that such a path can be extended to a solution path with exactly p edges without degrading the solution cost. This observation leads to our first pruning rule – a potential member T of L(j; k) is rejected if $T \geq X$ for some $X \in L(j; h)$ with $h < k$. Use of this rule now increases the worst-case complexity of the L(j; k) calculation to $O(p\, n\, r\, L_{max}^2)$. Nevertheless, in empirical testing the rule was found to eliminate a large number of paths, and as a result improved the overall execution time.

Further pruning is possible if we know the sum-max cost c of a feasible solution. Let T be any potential member of L(j; k), corresponding to subpath P. The component sum of T is a lower bound on the cost of any solution path using P. If the component sum of T exceeds c, we know that any solution involving P has cost exceeding c and must therefore be suboptimal. The vector T is then eliminated by this *forward* pruning rule.

Backward pruning is also possible. For any fixed j, the vectors W(i, j) increase monotonically in each component as i decreases. Thus if the sum of $W(i_0, j_0)$'s components exceeds c, then for all $i \leq i_0$ we know that (i, j_0) cannot be an edge in an optimal path. For every j we define low(c, j) to be the smallest index i such that the component sum of W(i, j) is less than c. Then in Step 2 of the construction algorithm above the lower bound of the i loop can be replaced with low(c, j). Backward pruning also allows us to free space used to represent path vector lists, an important consideration in extending the size of problems that can be solved (see Section 7).

As will be described in Section 7, the extended labeling algorithm (with the above pruning rules incorporated) has been successfully used on a number of nontrivial examples; in one case we solved a 1024 module, four-processor, four-stage problem, and in another case we solved a 128 module, sixteen-processor, eight-stage problem. Some problem classes (as the fluid flow problem mentioned in Section 1) tend to have a substantial number of modules, thus motivating the use of heuristics for the MLAA problem. Heuristics are also useful in providing feasible solutions for the forward and backward pruning rules, and are valuable even for the two-stage problem when n is so large that the $O(p\, n^2 \log n)$ complexity is daunting.

The heuristics to be developed here are built on the ability to solve the one-stage problem very efficiently, in O(p n log n) time. This efficient technique, based on the methodology developed in Nicol (1988), is now briefly described. First note that the minimum bottleneck value for the one-stage problem is identically one of $O(n^2)$ edge weights. A one-stage version of REACH establishes the feasibility of any candidate weight in O(p log n) time. The set of all edge weights x(i, j) can be conceptually organized in an upper triangular matrix, with i and j being the row and column indices, respectively. By Property 1, the entries in any row increase monotonically in the columns, while column entries decrease monotonically in the rows. The recursive bisection algorithm of Section 4 is then easily modified to find the minimum feasible edge weight using O(n) REACH calls, showing that the one-stage MLAA problem can be solved in O(p n log n) time.

Our approach to developing heuristics proceeds by *projecting* an r-stage problem onto a one-stage problem, which is then optimally solved. An optimal partition for the projected problem yields an approximate solution for the r-stage problem. It will also be seen that the ability to solve one-stage problems efficiently aids us in evaluating the quality of various heuristic solutions.

Two projections are quite natural. The first takes a module's r-stage weight vector and transforms it into a single-stage weight by summing all the weights. The second method transforms a module's weight vector by selecting the maximum component. The resulting methods are termed the *sum-projection* and the *max-projection* heuristics, respectively. The sum-projection heuristic is of particular interest here. As will be seen in Section 7, this heuristic produces better partitions in practice than the max-projection heuristic. Furthermore, it will now be shown how the quality of a partition it produces can be bounded, relative to the (unknown) optimal partition. This bound is particularly tight when the variance in module weights is small.

For i = 1, ..., p let W_i be the sum of execution times, at all stages, of all modules assigned to processor i under the partition produced by the sum-projection, with W_{max} the largest among these. Similarly, let V_i be the sum of all workloads on processor i under the optimal partition (with respect to the r-stage problem), and let V_{max} be the largest of these. By the optimality of the one-stage partition for the sum-projection, $W_{max} \le V_{max}$. Now for j = 1, ..., r let b_j be the bottleneck weight at stage j for the sum-projection partition, and let sp(j) denote the corresponding stage j bottleneck processor. Then for some positive $\alpha_j \leq 1$ we have $b_j = \alpha_j W_{sp(j)}$. The r-stage cost of the sum-projection partition is thus $C_{sp} = \sum_{j=1}^{r} \alpha_j W_{sp(j)}$, yielding

$$C_{sp} \leq \sum_{j=1}^{r} \alpha_j W_{max} \,. \tag{4}$$

Let C_{opt} be the cost of an optimal r-stage partition. Clearly $V_{max} \leq C_{opt}$, because V_{max} represents the work that must be done by one processor whereas C_{opt} is the total system finishing time. Thus we have $W_{max} \leq V_{max} \leq C_{opt}$, and consequently

$$W_{max}/C_{sp} \;\leq\; C_{opt}/C_{sp}. \tag{5}$$

Now we would like to develop bounds for the left hand side of this inequality.

For every module i, let $M^{(i)}$ and $m^{(i)}$ denote the respective maximum and minimum stage weights, with $\rho_i = m^{(i)}/M^{(i)}$ and $\rho^* = \min\{\rho_i : i = 1, \ldots, n\}$. Now α_j is the fraction of processor sp(j)'s total workload resident at stage j, and α_j is consequently maximized if all of the stage j module weights assigned to sp(j) are as large as possible, and if all of the other stage weights for sp(j) are as small as possible. Without loss of generality suppose that k modules are assigned to sp(j) and that sp(j) is processor one. We can then bound α_j from above using

$$\alpha_j \leq \frac{\sum_{i=1}^{k} M^{(i)}}{\sum_{i=1}^{k} \left[(r-1)m^{(i)} + M^{(i)} \right]} \,. \tag{6}$$

If $s_1, \ldots, s_k$ and $t_1, \ldots, t_k$ are nonnegative reals then a straightforward induction on k shows that

$$\frac{\sum_{i=1}^{k} s_i}{\sum_{i=1}^{k} t_i} \leq \max_{1 \leq i \leq k} \left\{ \frac{s_i}{t_i} \right\} .$$

Applying the above to inequality (6) produces

$$\alpha_j \le \max_{1 \le i \le k} \left\{ \frac{M^{(i)}}{(r-1)m^{(i)} + M^{(i)}} \right\} = \max_{1 \le i \le k} \left\{ \frac{1}{(r-1)\rho_i + 1} \right\} = \frac{1}{(r-1)\rho^* + 1} .$$

This allows us to bound C_{sp} in (4) from above, giving

$$C_{sp} \le \frac{r\, W_{max}}{(r-1)\rho^* + 1} . \tag{7}$$

Another bound on C_{sp} can be constructed in an analogous fashion. Let $S^{(j)}$ and $s^{(j)}$ denote (respectively) the maximum and minimum weights over all modules at stage j, with $\sigma = \sum_{j=1}^{r} s^{(j)}$. Then we have $\alpha_j \le \tau_j$, with

$$\tau_j = \frac{S^{(j)}}{\sigma - s^{(j)} + S^{(j)}} .$$

This yields the second bound

$$C_{sp} \le \tau W_{max} , \tag{8}$$

where $\tau = \sum_{j=1}^{r} \tau_j$.

Applying the above bounds (7), (8) on C_{sp} to (5) produces $\max\{1/\tau, [(r-1)\rho^* + 1]/r\} \le C_{opt}/C_{sp}$, which implies in particular that

$$\max\{1/\tau, \rho^*, 1/r\} \le C_{opt}/C_{sp} . \tag{9}$$

Relation (9) provides in turn an upper bound on how much the cost of the heuristic solution varies from the cost of the optimal solution, thus indicating the quality of the sum-projection heuristic.

The bounds above are especially encouraging when module weight variation, either within a module or within a stage, is known to be low. In the extreme case when a module's stage weights are all equal (so $\rho^* = 1$), or when all module weights within any stage are equal (so $\tau = 1$), relation (9) shows that $C_{sp} \le C_{opt}$ and thus the heuristic will in fact produce an optimal partition. In Section 7 we will encounter a problem class for which the value of ρ^* is no smaller than 1/2. We can then be assured that the sum-projection solution for this problem class is never more than twice as costly as the optimal solution. On the other hand, the $1/\tau$ bound is useful if any module weights are zero (forcing $\rho^* = 0$). One class of problems studied in Section 7 frequently has zero module weights.

The *a priori* bound (9) on solution quality is less satisfying when $1/\tau$, ρ^*, and $1/r$ are all small. An alternative *a posteriori* approach is based on our ability to quickly construct a lower bound on C_{opt}. Namely, any r-stage problem can be decomposed by stage into r separate one-stage problems. Let a_j be the optimal bottleneck value of the one-stage problem constructed from stage j module weights, and let opt_j be the stage j bottleneck value under an optimal partition for the r-stage problem. Clearly we must have $a_j \leq opt_j$ so that $\sum_{j=1}^{r} a_j \leq C_{opt}$. Given the cost C_h of the partition produced by some heuristic, the ratio C_{opt}/C_h is bounded from below by the ratio $\left(\sum_{j=1}^{r} a_j\right)/C_h$. Each a_j can be found in O(p n log n) time, making this a feasible mechanism for bounding the quality of any partition produced by a heuristic. Our experience with both exact and heuristic solution methods for r stages is summarized in the following section, together with computational evidence concerning the bounds on solution quality.

7. Computational Experience

We have implemented the extended labeling algorithm, as well as the two projection heuristics, in the C language on a networked, diskless SUN/3 workstation having 4 Mb of memory. This section describes the computational results obtained with these algorithms on three problem types. The performance benefits gained by using one of these algorithms as a preprocessor for a multiprocessor fluid flow code are also discussed.

In our study of the extended labeling algorithm, the number of modules (n) was chosen to be a power of 2. The upper limit on n varied by problem type, with the largest n attempted being 1024. In addition, p was chosen to be a power of two (between 2 and 16), and r was a power of two (between 2 and 8). Three classes of test problems were studied. In the first class, all module weights were randomly chosen from the uniform probability distribution on [1, 10001]. In the second class, weights for any given stage were created using a sine function over $[0, 2\pi]$ with amplitude 50, frequency 2, and translated upward in the y direction by the constant 150; all module weights are consequently between 100 and 200. To generate the weight for module i, we evaluate the sine function at $x = i \cdot 2\pi/(n-1)$ and retain the integer portion of that evaluation. The "phase constant" for such a curve is the smallest $x \in [0, 2\pi]$ at which the derivative is equal to 1. The curves for different stages have different phase constants: the phase constant for stage 1 is 0, and for stage $j > 1$ it is $\pi/2^{j-1}$. Figure 5 illustrates this weight generation procedure. The third class contains problems modeled on the fluid flow example described in Section 1. For this class we used at most four stages; these problems frequently have module weights of zero in regions where finer grids are absent.

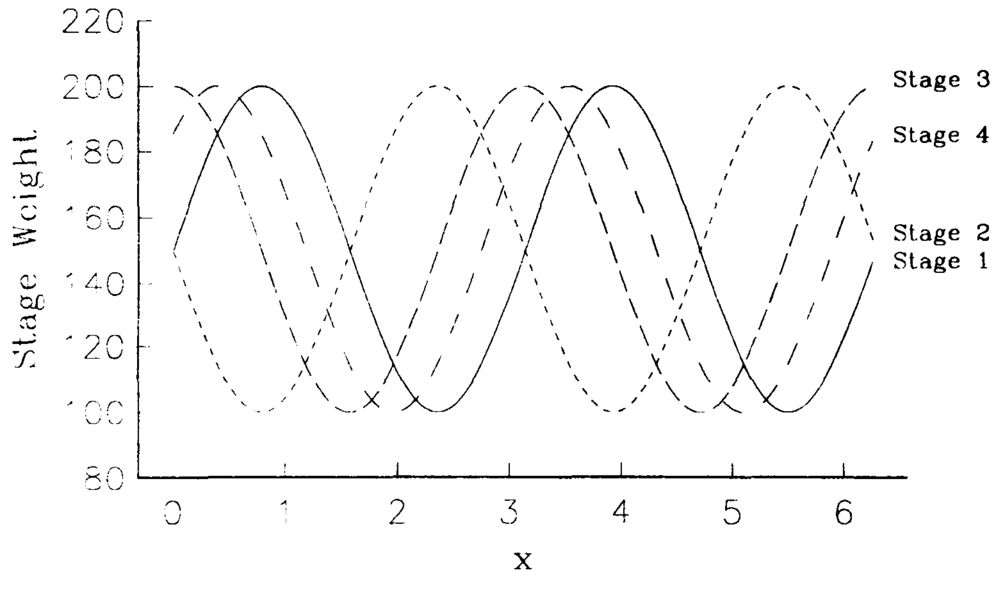

Figure 5. Module weights in a four-stage sine wave problem.

This computational study was devised to (i) determine how large a problem can be solved exactly, (ii) determine whether the refinements proposed improve our ability to solve larger problems, and (iii) assess the quality of solutions produced by the heuristics, when compared to the exact solution obtained by the labeling algorithm. In this empirical study, problems from a given class with identical size parameters behaved similarly. Consequently, we feel justified in reporting the results from a single (representative) run. In the tabulations that follow, two problems differing only in the number of processors had the same module weights.

Table 2 presents a summary of results obtained using the extended labeling algorithm (utilizing all refinements described in Section 6). Results for each problem class, indexed by the number of modules (n) and the number of processors (p), display three performance measures. The first measure is elapsed processing time for the problem, in seconds. This time excludes that required to generate a feasible solution using the sum-projection heuristic, as this amount was insignificant in comparison to the execution time required by the extended labeling algorithm. If the running time has an associated asterisk, that problem has four stages; otherwise it has eight stages. The second measure is the speed-up factor from using the refinements: namely, the ratio of processing time without refinements (the basic algorithm) to that utilizing the refinements. The third measure is the ratio of the maximum number of path vectors in use at any time while running the basic version, divided by the same quantity for the version with refinements. The maximum number of path vectors used is an important measure, as it quantifies the amount of storage needed by an algorithm. Indeed, for the system configuration used in conducting the tests, the limiting factor in solving the largest problems turned out to be the available storage rather than the computation time. The symbol (–) is used in Table 2 to indicate when the given problem could not be solved by the basic algorithm with the available resources.

Table 2 shows that for a variety of problem types the extended labeling algorithm can solve problems of reasonable size in a reasonable amount of time. These results also show that the refinements proposed can significantly reduce the algorithm's execution time and storage requirements, and in doing so increase the size of problems that can be solved exactly.

We are also interested in the quality of solutions produced by the proposed heuristics, and in the behavior of the a priori and a posteriori bounds on C_{opt}/C_{sp} developed in Section 6. We have applied the sum-projection and max-projection heuristics to each problem listed in Table 2 and have computed the ratio of the optimal cost divided by the heuristic cost. We have also computed the a priori and a posteriori bounds for the sum-projection heuristic. Table 3 displays the sample mean and standard deviation of the individual problem measurements for each problem class. For the problems studied here, the sum-projection always achieved a better solution than the max-projection, and it typically obtained solutions quite close to optimal. The sum-projection's a priori bounds tend to be low, due to significant variation in the module weights. On the other hand, the a posteriori bounds are reasonably tight, thereby justifying the additional effort needed to compute them.

The sum-projection heuristic has been used to produce partitions for a multiprocessor implementation of the fluid flow problem. The multiprocessor employed is a Flex/32, at the NASA Langley Research Center. The Flex/32 is a shared-memory architecture, with eighteen processors available for parallel processing. The grid problems studied consisted of 2048 coarse grid points and four stages; each problem was partitioned for sixteen processors. The flow problems studied tended to concentrate fine grids at one end of the domain. We compared the execution time of the sum-projection method against that of the simplest partitioning method, which simply divides the domain into sixteen equi-length pieces. On a suite of five such problems, computations partitioned using the sum-projection heuristic ran more than twice as fast compared to the equi-length partition. Data taken from these runs is presented in Table 4. This provides clear evidence that substantial performance benefits can be gained by using the MLAA solution techniques on real problems, implemented on real parallel architectures.

Table 2

Performance of the Extended Labeling Algorithm on Three Problem Classes

(Four-stage problems are marked with *, all others involve eight stages)

Uniform Random Problems

	p = 4			p = 8			p = 16		
n	time (secs)	time ratio	space ratio	time (secs)	time ratio	space ratio	time (secs)	time ratio	space ratio
32	3	3	3.5	8	6	6.1	156	–	–
64	14	1	3.6	99	6	5.9	286	–	–
128	10	3	3.4	2147	–	–	409*	–	–
256	24*	2	3.1	1139*	–	–			

Sine Wave Problems

	p = 4			p = 8			p = 16		
n	time (secs)	time ratio	space ratio	time (secs)	time ratio	space ratio	time (secs)	time ratio	space ratio
32	6	1	2.6	6	2	4.5	7	2	7.3
64	8	2	2.7	16	5.3	8.2	62	13	9.2
128	15	3	2.9	406	–	–	2960	–	–
256	190*	–	–						

Fluid Flow Grid Problems

	p = 4			p = 8			p = 16		
n	time (secs)	time ratio	space ratio	time (secs)	time ratio	space ratio	time (secs)	time ratio	space ratio
32	1*	3	1.8	1*	3	3.0	1*	4	3.6
64	4*	1	2.2	4*	2	4.0	4*	4	6.0
128	5*	4	3.3	17*	3	5.4	33*	6	5.3
256	21*	3	2.8	191*	–	–	232*	–	–
512	73*	35	3.1	5534*	–	–			
1024	4524*	–	–						

Table 3

Measurements and Lower Bounds on Ratio of Optimal Cost Divided by Heuristic Cost

	Uniform		Sine		Grid	
ratio	mean	deviation	mean	deviation	mean	deviation
max	0.92	0.04	0.94	0.20	0.79	0.11
sum	0.97	0.02	0.98	0.01	0.85	0.08
a priori	0.30	0.07	0.68	0.05	0.26	0.02
a posteriori	0.87	0.05	0.90	0.03	0.70	0.15

Table 4

Fluid Flow Problem Execution Times (in seconds)

run	serial time	MLAA time	MLAA speedup	standard time	standard speedup
1	149.9	14.9	10.0	32.8	4.5
2	142.1	14.3	9.9	30.8	4.6
3	130.2	13.4	9.7	30.0	4.3
4	110.2	12.5	8.8	28.4	3.8
5	101.6	11.7	8.7	22.9	4.4

8. Summary

This paper has considered the problem of optimally assigning modules to processors, so that the sum of maximum workloads at each stage is minimized. After formulation of this problem as a multi-objective network optimization problem, two polynomially-bounded algorithms were developed for the case of $r = 2$ stages. The general problem (for arbitrary r) was demonstrated to be NP-hard. In order to solve large-scale problems, two heuristic procedures were developed based on projecting an r-stage problem onto an efficiently solvable one-stage problem. Bounds on the solution quality produced by the heuristic methods were obtained using a priori and a posteriori information. Computational experience was presented with one of the exact algorithms, incorporating certain pruning rules and making use of a feasible solution produced by the sum-projection heuristic. The empirical results demonstrate that the exact method, suitably refined, can solve some reasonably large problems. For larger problems the sum-projection heuristic procedure shows a good deal of promise, exemplified by its success in partitioning processors for an actual fluid flow problem.

References

Berger, M. J. and J. Oliger. 1984. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *J. Comput. Physics* **53**, 484-512.

Berman, O., D. Einav and G. Handler. 1987. The Constrained Bottleneck Problem in Networks. Technical Report, College of Management, University of Massachusetts, Boston, Mass.

Bokhari, S. H. Partitioning Problems in Parallel, Pipelined and Distributed Computing. 1988. *IEEE Trans. on Computers* **37**, 48-57.

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

Hansen, P. 1980. Bicriterion Path Problems. In *Multiple Criteria Decision Making: Theory and Application*, G. Fandel and T. Gal (eds.). Springer-Verlag, Berlin, pp. 109-127.

Henig, M. 1985. The Shortest Path Problem with Two Objective Functions. *Eur. J. Opns. Res.* **25**, 281-291.

Kalaba, R. 1964. Graph Theory and Automatic Control. In *Applied Combinatorial Mathematics*, E. Beckenbach (ed.). Wiley, New York, pp. 237-252.

Kung, H. T., F. Luccio and F. P. Preparata. 1975. On Finding the Maxima of a Set of Vectors. *J. Assoc. Comput. Mach.* **22**, 469-476.

Nicol, D. 1988. Parallel Algorithms for Mapping Pipelined and Parallel Computations. ICASE Technical Report 88-2, NASA, Langley Research Center, Hampton, Va.

O'Hallaron, D. 1988. Computing the Cholesky Decomposition on the Warp Computer. 3rd International Conference on Supercomputing, Vol. 2, pp. 396-401.

Warburton, A. 1985. Worst Case Analysis of Greedy and Related Heuristics for Some Min-Max Combinatorial Optimization Problems. *Math. Programming* **33**, 234-241.

Warburton, A. 1987. Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems. *Opns. Res.* **35**, 70-79.

ORIGINAL PAGE IS
OF POOR QUALITY

NASA

Report Documentation Page

1. Report No. NASA CR-181748 ICASE Report No. 88-57	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle A MULTISTAGE LINEAR ARRAY ASSIGNMENT PROBLEM		5. Report Date November 1988
		6. Performing Organization Code
7. Author(s) D. M. Nicol, D. R. Shier, R. K. Kincaid, D. S. Richards		8. Performing Organization Report No. 88-57
		10. Work Unit No. 505-90-21-01
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-18107, NAS1-18605, AFOSR 88-0117
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to Operations Research

Final Report

16. Abstract

Implementation of certain algorithms on parallel computing architectures can involve partitioning contiguous elements into a fixed number of groups, each of which is to be handled by a single processor. It is desired to find an assignment of elements to processors that minimizes the sum of the maximum workloads experienced at each stage. This problem can be viewed as a multi-objective network optimization problem. Polynomially-bounded algorithms are developed for the case of two-stages, whereas the associated decision problem (for an arbitrary number of stages) is shown to be NP-complete. Heuristic procedures are therefore proposed and analyzed for the general problem. Computational experience with one of the exact problems, incorporating certain pruning rules, is presented for a variety of test problems. Empirical results also demonstrate that one of the heuristic procedures is especially effective in practice.

17. Key Words (Suggested by Author(s)) assignment, heuristics, multi-objective, network, parallel algorithms, workload balancing	18. Distribution Statement 61 - Computer Programming and Software 66 - Systems Analysis Unclassified - unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	33	A03