NASA Contractor Report 181655

ICASE REPORT NO. 88-2

ICASE

PARALLEL ALGORITHMS FOR MAPPING PIPELINED AND PARALLEL COMPUTATIONS

David M. Nicol

Contract No. NAS1-18107
April 1988

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Parallel Algorithms for Mapping Pipelined and Parallel Computations

*David M. Nicol**
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185

Abstract

Many computational problems in image processing, signal processing, and scientific computing are naturally structured for either pipelined or parallel computation. When mapping such problems onto a parallel architecture it is often necessary to aggregate an obvious problem decomposition. Even in this context the general mapping problem is known to be computationally intractable, but recent advances have been made in identifying classes of problems and architectures for which optimal solutions can be found in polynomial time. Among these, the mapping of pipelined or parallel computations onto linear array, shared memory, and host-satellite systems figures prominently. This paper extends that work first by showing how to improve existing serial mapping algorithms. Our improvements have significantly lower time and space complexities: in one case we reduce a published $O(nm^3)$ time algorithm for mapping m modules onto n processors to an $O(nm \log m)$ time complexity, and reduce its space requirements from $O(nm^2)$ to $O(m)$. We then reduce run-time complexity further with parallel mapping algorithms based on these improvements, that run on the architectures for which they creating mappings.

*This research was supported in part by the National Aeronautics and Space Administration under NASA contract NAS1-18107 while the author was in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

1 Introduction

Many computational problems in image processing, signal processing, and scientific computing are naturally structured for either pipelined or parallel computation. It is common for an obvious problem decomposition to have more components, or "modules" than there are processors. We must then map the computation by aggregating modules. The general mapping problem is known to be intractable but recently advances have been made by Bokhari[4] in identifying classes of problems and architectures for which optimal solutions can be found in polynomial time. Among these types of computations, a set of modules configured as a chain figures prominently. Unless otherwise stated, all references to Bokhari's work refer to [4].

As pointed out by Bokhari, the problem of mapping module chains onto different types of architectures frequently arises in image and signal processing applications; it may also arise in the parallel solution of partial differential equations. The concept of "module" can be quite general. For example, a signal processing application may require a signal to be Fourier-transformed, massaged in the frequency domain and then inverse-transformed. Each stage may be viewed as a module, or a stage may be subdivided into a sequence of modules. In an image processing context we may find similar processing stages for every frame of data. A common means of numerically solving a partial differential equation (PDE) in parallel is to decompose the PDE domain into strips[12]. The computation associated with a strip is the collection of all grid point updates required for points within the strip. The communication requirements between strips gives this computation a chain-like structure. At a given iteration, all strips may be updated in parallel, with communication occuring at the iteration's end. Grids may be irregular, giving strips different execution weights. A viable means of balancing the workload is to decompose the domain into many more strips than there are processors, and then aggregate them into equi-weighted superstrips. Modules are also easily identified in a computation described by a directed acyclic graph (DAG) whose nodes describe computations, and whose arcs define data dependencies. The "level" of a DAG node u is the smallest number of nodes on a path from any source node (no incoming arcs) to u; the collection of all nodes at a given level can constitute a module. It is not immediately obvious that a chain structure between modules should result, since a node in module k (equivalently, at level k) may depend on a node in module $i < k-1$. However, we can create a chain-like communication arrangement if we require every module j to transmit all its results to module $j+1$ and to transfer any results received from module $j-1$ which are to be used by modules $k > j$. We can then pipeline multiple independent invocations of the DAG computation.

Current parallel architectures compel us to at least consider chain decompositions. For example, the CMU Warp[1] is a linear array of high-powered processors, so that pipelining sequential modules is a natural solution approach. It can be advantageous to use chains even if the communication topology is rich. For example, the Intel iPSC (hypercube) has very high communication startup costs which are nearly independent of the message size.

Better performance is sometimes seen by minimizing the *number* of messages, rather than the message *volume* [14]. The performance-conscious programmer again is encouraged to limit the interconnection structure of the problem decomposition; a chain offers the simplest of useful structures.

Let $M_1, M_2, ..., M_m$ denote a chain of m modules which may be executed concurrently. As we have described, the modules may form a pipeline of computations or may describe a parallel computation whose communication requirements are local. The mapping problem under the *contiguity constraint* is to assign each M_i to one of n processors in such a way that the set of modules assigned to a processor forms a contiguous subchain of $M_1, \ldots, M_m$. The problem becomes non-trivial when we allow the modules to have individual execution times (called module weights), and require an explicit communication cost for mapping M_i and M_{i+1} onto different processors. A processor's time during the computation is spent either executing a module, communicating results, or waiting for results so that it can continue. Under any mapping there will be at least one "bottleneck" processor who limits the computational rate. We seek the mapping which minimizes the execution and communication time of the associated bottleneck processor. Bokhari gives polynomial-time algorithms for optimally mapping a chain onto a linear array of processors, mapping a chain onto a shared memory machine, and mapping a collection of chains onto a system consisting of a central host with a number of attached satellite processors.

Bokhari solves these problems with a layered graph. A graph node at layer i describes one possible assignment of modules to the *ith* processor. Layer i has a node for every possible assignment. Edges exist only between nodes in adjacent layers, and are always rooted in the layer with smaller index. An edge leaving a node is labeled with the cost of the associated processor assignment. Edges are defined so that every path through the graph describes a legal mapping, and the edges on that path can be analyzed to give the mapping's cost. A least-cost path algorithm is employed to find the optimal mapping. His algorithms map a chain onto a linear array in $O(nm^3)$ time and space, onto a shared memory machine in $O(nm^3 \log m)$[1] time and $O(nm^3)$ space, and map a set of n chains onto a host with n satellites in $O(nm^2 \log m)$ time and $O(nm^2)$ space.

The value of m can be quite large for applications whose modules are fine-grained. In such cases an $O(nm^3)$ algorithm is unattractive. This is especially true since the mathematical model we employ to assess a mapping's cost is quite simple, and ignores architectural details which may impact the accuracy of the model. Accepting that a simple mathematical model of the mapping problem is still desirable, it is important then to find ways to reduce the complexity of the approach. Iqbal[7] does so by considering approximation algorithms that find a solution guaranteed to be within ϵ of the true optimal. Letting W_T denote the sum of all module weights, his method finds the minimal approximate solution to the linear array problem in $O(nm \log(W_T/\epsilon))$ time, to the shared memory problem in $O(m^2 \log(W_T/\epsilon))$ time[2], and to the host-satellite problem in $O(nm \log(W_T/\epsilon))$ time.

[1] All logarithms in this paper are base 2.

[2] Iqbal incorrectly claims $O(m \log(W_T/\epsilon))$ for this solution.

These methods are attractive alternatives to Bokhari's, provided the user can accept the possibility of failing to find the precise optimal solution. Another drawback is that the complexity of Iqbal's method is sensitive to the actual values of the module weights and on the degree of accuracy desired.

Bokhari's methods and Iqbal's methods both rely on a "probe" function which finds an optimal solution, subject to some constraint. The probe function is repeatedly called, varying the the constraint, until an optimal solution is discovered. In § 3 we outline Bokhari's solutions, and show how they are all easily improved by a factor of m by reducing the complexity of his probe function. We then examine each problem, and show how to reduce the complexities of their respective probe functions, how to reduce the cost of organizing the set of probe calls, and how to achieve low expected parallel time complexities by executing the mapping algorithms on the target architectures. These algorithms' expected complexities are based on the assumption that all module weights are independent samples of a common unspecified distribution, and that all communication delays are independent samples of a different unspecified distribution.

In § 4 we reduce the time complexity of Iqbal's probe function from $O(nm)$ to $O(n \log m)$. The improvement requires only the additional assumption that communication costs are bounded. Then we exploit the problem's structure and reduce the cost of organizing the probe search values from $O(m^2 \log m)$ to $O(m \log m)$. The resulting algorithm has $O(nm \log m)$ time complexity and $O(m)$ space complexity. Finally, we organize the algorithm for execution on the linear array itself. The parallel algorithm has an $O(m \log m \log n)$ time complexity, and $O(nm)$ space complexity.

In § 5 we reduce the time complexity of a probe function based on Kernighan's algorithm[8] from $O(m^2)$ to $O(m \log m)$. Coupled with the search organization developed for the linear array problem, we reduce this problem's time complexity from Bokhari's $O(nm^3 \log m)$ to $O(m^2 \log m)$. Our algorithm has $O(m^2)$ space complexity. We then parallelize our solution in three ways. One method achieves an $O((m^2/n) \log m \log n)$ time and $O(nm)$ space complexity; a second achieves an $O((m^2/n) \log m)$ time and $O(m^2)$ space complexity. The third is appropriate when $8n^3 < m$, and has an expected $O((m^2/n) \log m)$ time and $O(m)$ space complexity.

Finally, in § 6 we use the results of § 4 to reduce the solution time complexity from Bokhari's $O(nm^2 \log m)$ to an $O(\max\{nm \log n, n \log^2 m\})$ time complexity. Our algorithm has $O(nm)$ space complexity. We then parallelize the algorithm for execution on the host-satellite architecture. When m is sufficiently larger than n the parallel algorithm has an $O(nm)$ time complexity which is within a constant factor of optimal when the problem is loaded serially.

The trade-offs between communication costs and load balance have recently been addressed by a few researchers. Berger and Bokhari in [2] propose and analyze binary dissection of a two-dimensional domain with irregular workload. The solutions they construct need not be optimal. A similar problem for finite-element solution methods was studied by Sadayappan and Ercal in [13]. Cventanovic in [6] examines mapping, communication,

and grainularity issues in an abstract setting. Foundational work for the *parallel* mapping problem was laid by study of the *distributed* mapping problem. The seminal works in this field include papers by Stone [17],[16]; by Bokhari [5], and by Towsley [18]. Bokhari summarizes much of this work in [3].

2 Model Definitions

We suppose that a computational problem has been decomposed into m *modules* $M_1, \ldots, M_m$. These modules may be defined by function, e.g. fast fourier transform, convolution; they may also be some partition of a data domain, as in the solution of partial differential equations. We imagine that one execution of M_i needs data from M_{i-1}, M_{i+1}, or both. We suppose that a module M_i will be executed many times, each execution requiring $w_i > 0$ time on one of a set of n homogeneous processors; the modules are concurrent because either results are being pipelined, or the modules are loosely synchronized and exchange the necessary data at the conclusion of every iteration. Our expected complexity analysis will assume that each w_i is drawn independently from a common distribution having finite mean μ_w and standard deviation σ_w.

We are interested primarily in situations where m is large and $n \ll m$, e.g., $n = 10$ and $m = 1000$. One reason for this focus is that algorithms we develop are somewhat more complex than existing ones; for small m the existing algorithms are likely to be fast enough for practical use; conversely, for large m they are impractical. A second reason is that using the parallel processors to compute the mapping of another computation imposes additional overhead, and becomes an attractive option only if the problem size is large enough to overcome that overhead.

At the end of M_i's execution period there is data available for consumption by M_{i-1} and/or M_{i+1}. If one of these modules (say M_{i-1}) is assigned to the same processor as M_i, we assume that the next invocation of M_{i-1} can access that data without additional cost. If one of these modules (say M_{i+1}) is assigned to a different processor, then M_i's processor must explicitly send data over a communication channel, and we will say that the logical link between M_i and M_{i+1} is *exposed*. The cost of that communication is assumed to depend on the communicating modules. Exposing the link between M_i and M_{i+1} causes both modules to incur a delay cost $C_i \geq 0$ which models all overhead a processor suffers in sending and receiving messages over that link. We make the reasonable assumption that $C_i < \mathcal{C}$ for some constant $\mathcal{C}$ which is independent of m. Our expected complexity analysis assumes that each C_i is drawn independently from a common distribution having finite mean μ_c and standard deviation σ_c.

We let $S_{ij} = \sum_{k=i}^{j} w_k$ denote the sum of module weights on the subchain delimited by M_i and M_j. S_{ij} is a single processor's *module evaluation time* cost of being assigned the subchain. The incorporation of the associated delay costs C_{i-1} and C_j will depend on the architecture considered, as shown below.

Linear Array: Consider a linear array of processors $P_1, P_2, \ldots, P_n$; processor P_i has a direct communication link with only processors P_{i-1} and P_{i+1}. We assume that a processor is not free to proceed with computation if it is actively engaged in communication. If modules M_i through M_j are assigned to processor P_k, then P_k's *execution time* during one iteration is $C_{i-1} + S_{ij} + C_j$. The cost of a complete mapping is the maximum processor execution time among all processors; we have called the processor defining this maximum the bottleneck processor. If the system we map is a pipeline, then the bottleneck processor limits the rate of results, and the mapping cost is the time required to obtain one result from a full pipeline. If the system is parallel rather than pipelined the mapping cost is the time required by each iteration. In either case we optimize performance by minimizing the mapping cost.

Shared Memory Machine: Consider a collection of identical processors that communicate through a shared memory. The communication medium is a shared resource, so that it is appropriate to model communication overhead by adding the costs of all exposed links. The cost of a mapping is the maximum of (i) the sum of all communication costs on exposed links, and (ii) the maximum processor module evaluation costs under the assignment. This model presumes that communication can be overlapped with computation, but that the communication medium serializes the communication traffic.

Host-Satellite Machine: Consider a powerful host machine which has n satellite processors. This arrangement might be appropriate when there are n sensors with attached micro-processors. There is a chain-like pipelined computation associated with each satellite. Without loss of generality we assume that the chain for satellite P_i has m modules, $M_{i1}, \ldots, M_{im}$. Satellite P_i can unload some subchain M_{ij} through M_{im} onto the host at the cost of an inter-module communication C_{ij} which is suffered by both host and satellite (keeping the whole subchain on the satellite gives a communication cost $C_{i(m+1)}$). Unloading work onto the host also has the effect of increasing the host's computational load. The host's cost of a mapping is the sum of (i) any load it must always perform, e.g. combination of fully processed sensor data, (ii) the sum of module execution times of all satellite modules it has received, and (iii) the communication costs associated with each satellite. A satellite's execution time is its module evaluation costs plus its host communication cost. An assignment's cost is the maximum of host cost and maximal satellite execution cost.

The following section sketches Bokhari's approach to solving these mapping problems, and points out an easy improvement to his algorithms.

3 Layered Graph Path Algorithms

Bokhari solves the linear array problem by finding the minimum path through a specially created layered graph. The graph has a source node $< s >$ and a sink node $< t >$. Each layer corresponds to a processor. Layer i contains a node for every legal means of assigning modules to processor i. For example, node $< j, k >$ at layer i represents the assignment

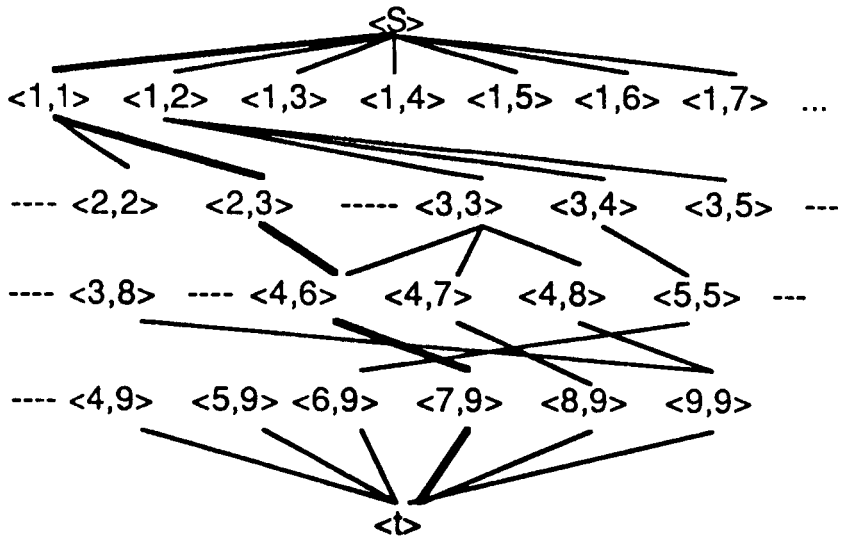

Figure 1: Layered Graph for Linear Array Problem, 9 modules, 4 processors

of modules M_j through M_k to processor i. Each layer contains $O(m^2)$ nodes. An edge is directed from node $< j, k >$ in layer i to any node of the form $< k+1, l >$ in layer $i+1$. $< s >$ directs an edge to every node at layer 1, and every node in layer n directs an edge to $< t >$. Consequently, any path from $< s >$ to $< t >$ corresponds to an assignment which satisfies the contiguity constraint. Figure 1 illustrates Bokhari's own example; while an assignment path is shown, many edges are not shown in order to relieve visual congestion. The layered graph assumes that every processor receives at least one module.

An edge out of node $< j, k >$ at layer i is labeled with the value $C_{j-1} + S_{jk} + C_k$. It is possible to include a dependence on i here to model heterogeneous processors and communication links; for simplicity we assume homogeneity. The cost of a path is the value of the maximally weighted edge on the path, which clearly is the time required by the bottleneck processor to solve its portion of the problem. A standard least-cost path algorithm finds the optimal mapping in $O(\textit{graph edges})$ time, in this case $O(nm^3)$.

Least-cost paths through layered graphs are also at the heart of Bokhari's shared-memory and host-satellite problem solutions. Here he develops a general technique of analyzing *Sum-Bottleneck* graphs. An edge e on such a graph has a *sum-weight* and a *bottleneck-weight*. The cost of a given path through the graph is the maximum of (i) the sum of all sum-weights on the path's edges, and (ii) the maximum bottleneck-weight among the path's edges. The path with minimal cost is found by first identifying all

unique bottleneck-weight values, and by sorting them. Then a binary search on the list of bottleneck-weight values is performed—for each bottleneck-weight value b visited, a shortest path routine TESTPATH(b) is called. TESTPATH(b) treats any edge whose bottleneck-weight value is greater than b as non-existent. If there is a path from source to sink on this edge-reduced graph, then TESTPATH(b) returns the path whose sum of sum-weights is minimal. If there is no path between source and sink TESTPATH(b) returns the null path whose cost is defined to be ∞. Defining $S(b)$ to be the length of the path returned by TESTPATH(b), the binary search seeks the smallest bottleneck value $\hat{b}$ such that $\hat{b} \geq S(b)$. The optimal sum-bottleneck solution is then either $\hat{b}$ or $S(\tilde{b})$, where $\tilde{b}$ is the greatest bottleneck value less than $\hat{b}$. For each of the layered graphs considered a call to TESTPATH(b) has complexity $O(\textit{graph edges})$.

The sum-bottleneck graph for the shared-memory problem is topologically equivalent to that for the linear array problem. An edge directed out of node $< j, k >$ is labeled with bottleneck-weight S_{jk} and sum-weight C_k. Each call to TESTPATH has complexity $O(nm^3)$; the algorithm's $O(nm^3 \log m)$ complexity follows from the observation that there are $O(m^2)$ unique bottleneck values, and hence $O(\log m)$ calls to TESTPATH.

The sum-bottleneck graph for the host-satellite problem again associates a layer with a processor. Node $< j >$ at layer i represents the mapping of satellite P_i's first j modules onto the satellite, with the remaining modules being mapped onto the host. A node at layer i directs an edge to every node at layer $i + 1$. An example of this graph is shown in figure 2. The bottleneck weight on an edge directed out of node $< j >$ in layer i is the sum of weights of modules M_{i1} through M_{ij}, plus the communication cost C_{ij}. The sum weight on that edge is the sum of $M_{i(j+1)}$ through M_{im} weights with the communication cost C_{ij}. To account for an initial host load H, every edge directed out of the source node has a sum weight of H and a bottleneck weight of zero. Each call to TESTPATH has $O(nm^2)$ time complexity. There are possibly nm unique bottleneck values, giving a $O(nm^2 \log m)$ overall complexity.

The least-cost path algorithm underlying these solutions exploits the fact that the graph is layered—for node v at layer i, the least-cost path from the source to v, through node u at layer $i - 1$ must include the least-cost path from the source to u. In fact, this is just a statement of the principle of optimality. The algorithm finds the least-cost paths from the source to all nodes at layer $i - 1$ before computing any least-cost path to a node at layer i. The least-cost path to v is found by examining every u which directs an edge to v and then extending the least-cost path to u with the $u \rightarrow v$ edge. The least-cost extension is the least-cost path to v. As we have previously stated, the complexity of this approach is proportional to the number of graph edges. A simple trick will reduce the number of graph edges without affecting path costs. For the linear array and shared memory problem graphs we add $n - 2$ layers, one between each of the previous layers (except between layers 1 and 2 where an additional layer provides no benefit). Each new layer has m nodes, labeled 1 through m. To avoid confusion we will refer to the "ith" layer in the new graph as being identical to the ith layer in the original graph. Node $< j, k >$ in layer i directs a

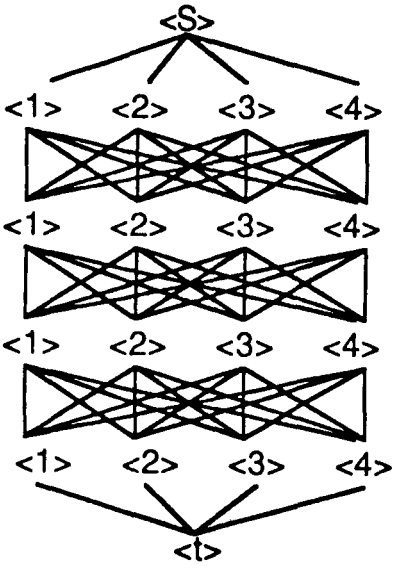

Figure 2: Layered Graph for the Host-Satellite Problem

single edge to node $< k >$ in the new layer between layers i and $i+1$; this edge is labeled exactly as before. Node $< k >$ in the new layer in turn directs an edge to every node of the form $< k+1, l >$ in layer $i+1$; every such edge is labeled with weight zero. Figure 3 illustrates the new graph. Again, many nodes and edges are not shown in order to avoid congestion. It is clear that any path from source to sink still defines a legal assignment and has a weight identical to that of the corresponding path in the original graph. The number of edges drops from $O(nm^3)$ to $O(nm^2)$, reducing the complexity of both the linear array and shared memory problems by a factor of m.

We treat the host-satellite assignment graph similarly. Between layers we interpose a single node. Every node at layer i directs a single edge to the node between layers i and $i+1$; the edge is weighted as before. The node between layers i and $i+1$ directs an edge to every node in layer $i+1$. The two weights on each such edge are zero. Once again, every path identifies an assignment and its cost; by reducing the number of graph edges by an order of m we reduce the algorithm's cost by an order of m. This same trick can be applied to the algorithms in [5] and [18][3].

Bokhari does not discuss parallelization of his methods on the target architectures. Even after improvement, his linear array solution is very ill-suited for parallelization on the array. A natural approach is to partition the solution graph, and require every proces-

[3]Private communication from Shahid Bokhari

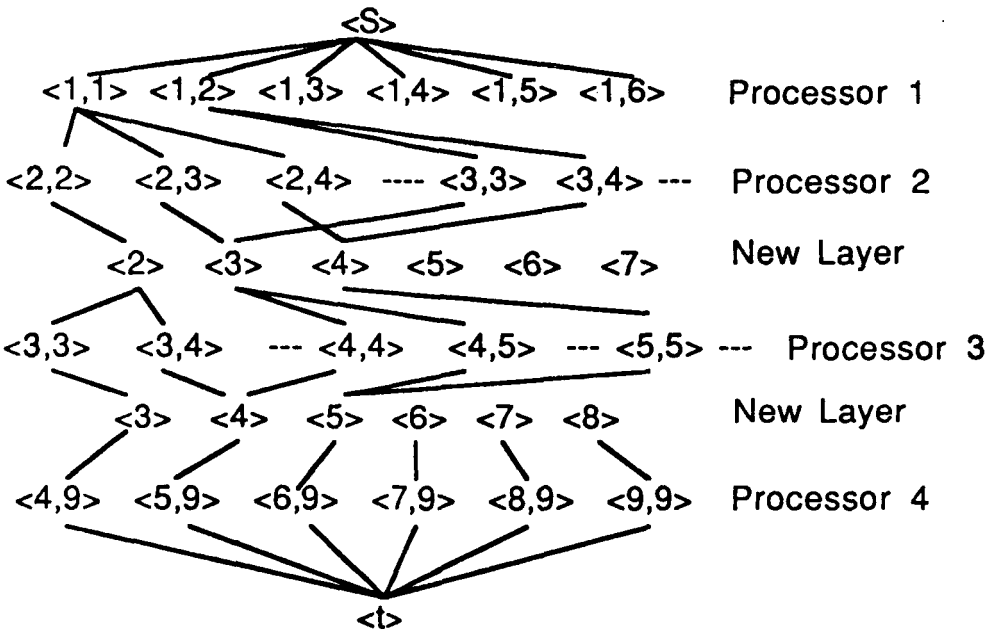

Figure 3: Improved Layered Graph for Linear Array Problem, 9 modules, 4 processors

sor to compute the least cost path to the nodes it is assigned. The computation proceeds in stages—find the least-cost paths to layer 2 nodes, then layer 3 nodes, etc. It is not difficult to see however that the communication requirements of this approach are enormous: there is communication across at least one link for every graph edge cut by the partition. Furthermore, if the nodes are distributed evenly among processors, then $\Omega(m^2)$ values will have to be broadcast between each of $n-1$ steps. The communication complexity alone is equivalent to the complexity of a serial solution. The method just described might work well on a shared memory machine, provided that the number of processors is small, and that the communication network is fast relative to the processor speeds. The cost model assumes serialized communication, so again we have an $O(nm^2)$ communication complexity. These observations also apply to a host-satellite system if the satellites in a host-satellite system can communicate through the host's memory.

If we have a computation which is decomposed into a very large number of modules, and if we desire to take advantage of the parallel hardware our mapping methods target, then Bokhari's methods leave room for improvement. In the following sections we discuss improved serial algorithms, and give parallel mapping algorithms based on these improvements.

4 Linear Array Problem

Bokhari's method for solving the linear array problem does not rely on a probe in the same way that his shared-memory and host-satellite solutions do. Our approach is based on Iqbal's[7], who developed a probing approach for finding an approximate solution. Like Bokhari's sum-bottleneck method we will probe the space of bottleneck values. Our improvements stem from increasing the efficiency of the probe method, and from exploiting the problem structure to avoid the cost of sorting all bottleneck values. The subsections to follow discuss these improvements, show how to parallelize the algorithm for execution on the linear array.

4.1 An Improved Probe Function

Our method is based on Iqbal's probe function PROBE1(w), which is shown in figure 4. PROBE1(w) determines whether it is possible to assign the workload so that every processor's execution time is less than or equal to the *bottleneck constraint* w. PROBE1(w) iteratively chooses a feasible subchain load for the "next" processor. Given that a processor's subchain begins with module M_i, PROBE1(w) finds that j such that (i) $\Omega_{ij} = C_{i-1} + S_{ij} + C_j \leq w$, and (ii) the remaining unassigned load $\Delta_j = C_j + S_{(j+1)m}$ is minimized. Iqbal proves that this rule will find an assignment whose cost is no greater than w, if one exists.

In the worst case, for every processor assignment PROBE1(w) will consider making module M_j ($j > n$) a subchain right endpoint. PROBE1(w) *always* considers making M_j an endpoint on every iteration where M_j is still unassigned. This gives PROBE1(w) an $O(nm)$ complexity.

Consider the problem faced by the inner loop of PROBE1(w): among all $j \in [i, m]$ such that $\Omega_{ij} \leq w$, find the $j_{\min}$ minimizing Δ_j. PROBE1 examines the entire interval $[i, m]$ for this point; instead we appeal to the problem's structure and quickly find a small subinterval $[k_{\min}, k_{\max}]$ which must contain $j_{\min}$.

Define the functions $\Delta_j^{-1} = S_{1j} + C_j$, and $w(i) = w - C_{i-1} + S_{1(i-1)}$ and note that

$$C_{i-1} + S_{ij} + C_j \leq w \quad \Leftrightarrow \quad S_{1(i-1)} + S_{ij} + C_j \leq w - C_{i-1} + S_{1(i-1)}$$

or equivalently,

$$\Omega_{ij} \leq w \quad \Leftrightarrow \quad \Delta_j^{-1} \leq w(i).$$

If we can find the largest j such that $\Delta_j^{-1} \leq w(i)$ we will have found the largest j such that $\Omega_{ij} \leq w$. Let $k_{\max}$ denote this upper bound. $k_{\max}$ can be quickly found with a pre-computed array *right_min*, whose jth entry equals k if the minimum value of Δ^{-1} over $[j, m]$ occurs at position k. *right_min* is computed once in $O(m)$ time, and is thereafter employed by every probe call. $\Delta^{-1}_{right\_min(j)}$ necessarily increases monotonically in j. Given w and i, $k_{\max}$ is simply the greatest index j greater than or equal to i such that $\Delta^{-1}_{right\_min(j)} \leq w(i)$. If

Definitions

W_T	Sum of all modules weights: $W_T = \sum_{i=1}^{m} w_i$
Ω_{ij}	Processor cost if assigned subchain $M_i, \ldots M_j$
	$\Omega_{ij} = C_{i-1} + S_{ij} + C_j$;
Δ_j	Total "remaining" load after assigning M_j: $\Delta_j = C_j + \sum_{k=j+1}^{m} w_j$

```
function PROBE1 (w) :Boolean;
    {
        i = 1; p = 1; k = 0; Δmin = W_T;
        while p ≤ n do
            {
                for j = i to m do
                    if Ω_ij ≤ w and Δ_j < Δmin then
                        {
                            Δmin = Δ_j;
                            k = j;
                        }
                Assign subchain M_i, ..., M_k to processor p;
                if k = m then return(true);
                i = k + 1; p = p + 1;
            }
        return(false);
    }
```

Figure 4: Iqbal's probe function

$k_{\max}$ exists, it can be found in $O(\log m)$ time with a binary search. If the search fails to find a feasible solution then no solution exists.

Having found $k_{\max}$ we can find the lower bound $k_{\min}$. Note first that

$$\Delta_{k_{\max}} = C_{k_{\max}} + S_{(k_{\max}+1)m}.$$

As j decreases, $S_{jk_{\max}}$ necessarily increases, and eventually exceeds $C_{k_{\max}}$. We choose $k_{\min}$

to be the largest j where this occurs. For any $j < k_{\min}$ we have

$$\Delta_j = C_j + S_{(j+1)k_{\max}} + S_{(k_{\max}+1)m} > C_{k_{\max}} + S_{(k_{\max}+1)m} = \Delta_{k_{\max}}.$$

Consequently, any $j < k_{\min}$ may be ignored as a solution. If $k_{\min} < i$, we take $k_{\min} = i$.

Since $S_{jk_{\max}}$ must increase as j decreases, $k_{\min}$ can be found with another binary search, on the "virtual array" $S_{ik_{\max}}, \ldots, S_{k_{\max}k_{\max}}$. Note that for any ij, $S_{ij} = S_{1j} - S_{1(i-1)}$, so that S_{ij} can be computed in constant time if the S_{1k}'s are pre-computed. This means that the virtual array need not be explicitly computed, and the search for $k_{\min}$ requires only $O(\log m)$ time.

A linear scan for feasible points in $[k_{\min}, k_{\max}]$ will find the feasible point minimizing Δ. Since we have assumed that the communication costs are bounded from above by some constant independent of m, the linear scan takes $O(1)$ time. Figure 5 presents pseudo-code describing this new $O(n \log m)$ probe function PROBE2(w). Note that a returned value of false occurs only if for some processor there are no feasible assignments. Like Iqbal's probe, PROBE2 will return true if a feasible mapping is found which uses fewer than n processors.

4.2 Improved Search Organization

At this point we could simply sort the $O(m^2)$ unique processor loads, and find the smallest feasible one with $O(\log m)$ calls to PROBE2. This algorithm's complexity is dominated by the $O(m^2 \log m)$ complexity of sorting. To further improve the probing approach we will have to reduce the cost of organizing the search. We do so by replacing the $O(m^2 \log m)$ cost of finding $O(\log m)$ probe values with an $O(m \log m)$ cost of finding $O(m)$ probe values. Because the probe calls are cheap, increasing their frequency to avoid a sort improves the overall performance.

For the moment, assume that all communication costs are zero so that every processor's execution time is of the form S_{ij}. Furthermore, we extend the definition of S_{ij} to allow $i > j$:

$$S_{ij} = S_{1j} - S_{1(i-1)}.$$

This definition encompasses the earlier one, and also shows that S_{ij} can be computed in constant time if all sums of the form S_{1k} are known.

We are able to infer that some execution time weights are larger than others, regardless of the module weight values. In particular, $S_{ij} < S_{ik}$ whenever $j < k$, and $S_{ij} > S_{kj}$ whenever $i < k$. This partial ordering is illustrated in figure 6 with a *dominance matrix.* Row entries ascend in value from left to right, column entries descend from top to bottom. By transitivity it follows that $S_{ij} < S_{uv}$ whenever $i \geq u$ and $j \leq v$.

We will call any contiguous portion of a row a *strip.* On any given strip we can use binary search and a probe function to identify the entry with smallest execution time weight that satisfies the probe. This observation allows us to eliminate large portions of

```
function PROBE2 (w) :Boolean;
    {
        i = 1; p = 1; k = 0;
        while p ≤ n do
            {
                Δ_min = W_T;
                w(i) = w − C_{i−1} + S_{1i};
                Use binary search to find k_max: the greatest j
                    such that Δ_{right_min(j)} ≤ w(i);
                If no such k_max exists return(false);
                Use binary search to find k_min: the greatest j ≤ k_max
                    such that S_{j k_max} ≥ C_{k_max};
                for j = k_min to k_max do
                    if Ω_ij ≤ w and Δ_j < Δ_min then
                        {
                            Δ_min = Δ_j;
                            k = j;
                        }
                Assign subchain M_i, ..., M_k to processor p;
                if k = m then return(true);
                i = k + 1; p = p + 1;
            }
    }
```

Figure 5: Improved Probe Function for Linear Array Problem

the search space. Consider a rectangular region of the dominance matrix that is h entries high and l entries long. Consider the effect of doing a binary search on the strip which best bisects the rectangle into equal sized pieces. Let S_{ij} be the minimal feasible strip entry found by the search. Any S_{uv} with $u \leq i$ and $v \geq j$ lies above and to the right of S_{ij}; any such entry dominates S_{ij} and may therefore be discarded as a solution possibility. Any S_{xy} with $x \geq i$ and $y < j$ lies below and to the left of S_{ij}; any such entry is dominated by the value $S_{i(j-1)}$ which is known to have failed. Such entries may also be discarded as a solution possibility. Since the strip bisects the rectangle into equal sized pieces, one half of the rectangle's entries are eliminated by the binary search; the remaining entries fall into no more than two regions which are again rectangular. These points are illustrated graphically in Figure 7. In order to find the minimal feasible solution within the rectangle

S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	S_{18}	S_{19}
S_{21}	S_{22}	S_{23}	S_{24}	S_{25}	S_{26}	S_{27}	S_{28}	S_{29}
S_{31}	S_{32}	S_{33}	S_{34}	S_{35}	S_{36}	S_{37}	S_{38}	S_{39}
S_{41}	S_{42}	S_{43}	S_{44}	S_{45}	S_{46}	S_{47}	S_{48}	S_{49}
S_{51}	S_{52}	S_{53}	S_{54}	S_{55}	S_{56}	S_{57}	S_{58}	S_{59}
S_{61}	S_{62}	S_{63}	S_{64}	S_{65}	S_{66}	S_{67}	S_{68}	S_{69}
S_{71}	S_{72}	S_{73}	S_{74}	S_{75}	S_{76}	S_{77}	S_{78}	S_{79}
S_{81}	S_{82}	S_{83}	S_{84}	S_{85}	S_{86}	S_{87}	S_{88}	S_{89}
S_{91}	S_{92}	S_{93}	S_{94}	S_{95}	S_{96}	S_{97}	S_{98}	S_{99}

Figure 6: Dominance Matrix of S_{ij} values

it suffices to apply this procedure recursively to the remaining rectangles. The recursion stops when a rectangular region consists only of a strip; then a binary search finds the best feasible strip solution, if one exists.

The efficiency is enhanced if throughout the search we maintain variables V_f and V_s. V_f records the largest execution time tested so far which failed the probe test, V_s records the smallest execution time tested so far which satisfies the probe. If the search procedure calls for a value V to be tested, the probe function needs to be called only if $V_f < V < V_s$. If the probe is called, either V_f or V_s will be updated, depending on the probe outcome. At the end of the search procedure V_s contains the minimal mapping cost. If the associated mapping has not been saved, a last call to PROBE2 will create it.

The lattice search technique calls the probe function more often than a binary search over a fully sorted set of bottleneck values, but avoids the high cost of sorting that set. Its utility rests in that it calls the probe function only $O(m)$ times, a fact we now demonstrate.

Define a *rectangle evaluation* to be the process of choosing a strip on a given rectangle, finding the minimum strip value satisfying PROBE2 (if any), and identifying the smaller rectangles, called *children*, which must also be evaluated. It is helpful to view the search process as a sequence of steps, where step 0 is the initial rectangle evaluation on the entire matrix. Step 1 consists of evaluating all children of step 0. In general, the *ith* step is composed of all evaluations of children defined by the previous step. We will say that a matrix entry is *active* at the beginning of the *ith* step if it lies within some rectangle that is evaluated during the *ith* step. We will also say that an entry is *evaluated* during the *ith* step if it lies on a strip over which a binary search occurs during the *ith* step. An evaluated entry need not actually be touched by the search. Three observations are key.

- The number of active entries in any matrix column decreases by one half every step.
- The total number of evaluated entries during any step is no greater than m.

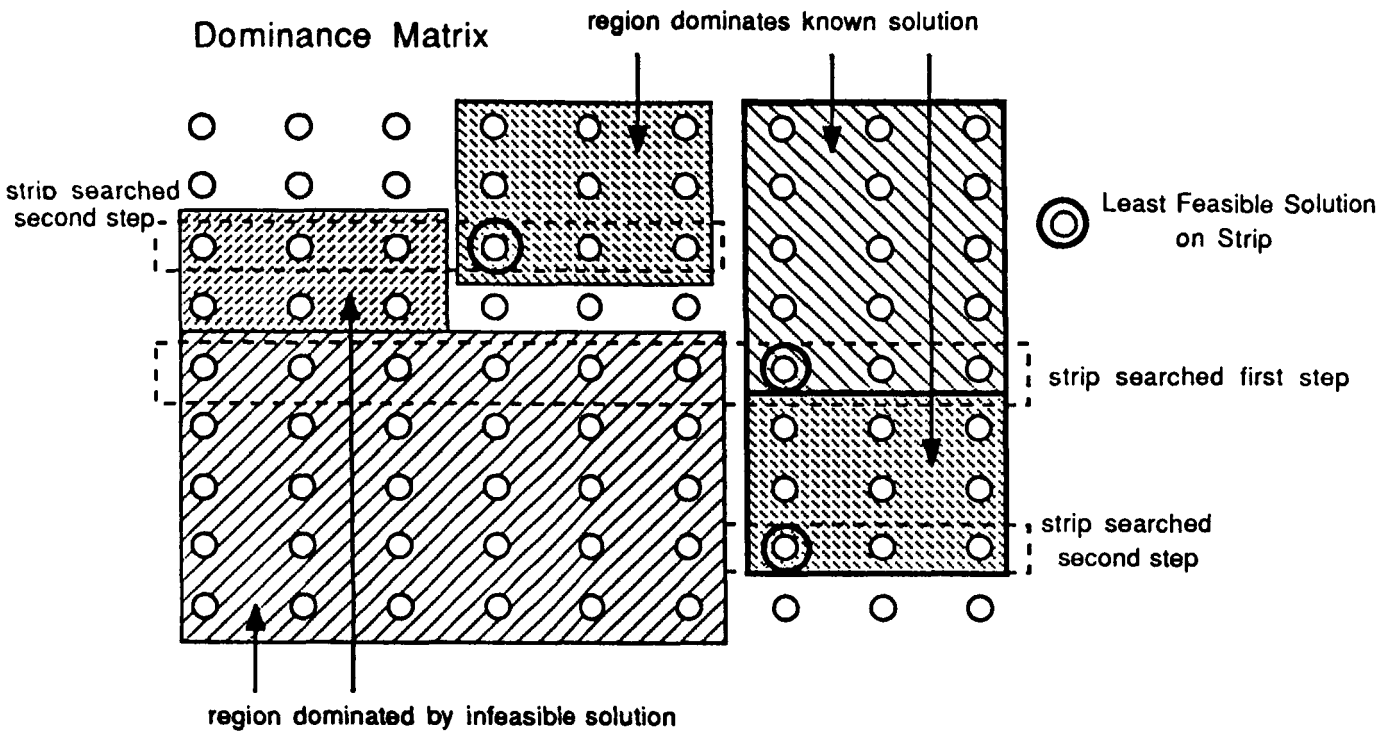

Figure 7: Lattice Search Method

- The maximum number of rectangles which are evaluated at step i is 2^i.

To see that the first point is true, consider any column in an evaluated rectangle. If the point found by the binary search lies in the column, or in one to the left, then only the lower half of the column entries are left active. If the point lies to the column's right, then only the upper half of the column's entries are left active. The second point follows from the observation that during a step, no two evaluated rectangles overlap in any row or column coordinates. If we sum the horizontal lengths of all evaluated rectangles the result is exactly m. The third point is obvious, since any rectangle evaluation spawns no more than 2 children.

From the first point we infer that there are no more than $\log m$ steps in the search. The number of PROBE2 calls required is the sum of calls by the binary searches involved. Because of the concavity of the log operation, the number of calls at a step is maximized when there are as many binary searches as possible, over short lists. A binary search on a list of k items requires no more than $\log k + 1$ probes. There are no more than m evaluated points at a step, and no more than 2^i binary searches. The number of probe calls at a step is consequently bounded from above by $2^i(\log(m/2^i) + 1)$. By summing over all steps, we

find the number of PROBE2 evaluations to be bounded by

$$\begin{aligned}\sum_{i=0}^{\log m} 2^i(\log(m/2^i)+1) &= \log m \sum_{i=0}^{\log m} 2^i - \sum_{i=0}^{\log m} i2^i + \sum_{i=0}^{\log m} 2^i \\ &< 4m.\end{aligned}$$

The evaluation of $\sum_{i=0}^{\log m} i2^i$ is accomplished using a general formula found in [9]. At the cost of adopting $O(m)$ probe calls, we avoid the cost of a full sort. There is a payoff. $O(m)$ calls to an $O(n \log m)$ probe gives an $O(nm \log m)$ algorithm, over the $O(m^2 \log m)$ alternative.

This search technique relies heavily on the lattice-like partial ordering of the dominance matrix. Redefining the dominance matrix by replacing each S_{ij} with $\Omega_{ij} = C_{i-1} + S_{ij} + C_j$ destroys that partial ordering. However, a similar ordering can be discovered in $O(m \log m)$ time with the following observation:

$$\begin{aligned}C_0 + S_{1j} + C_j < C_0 + S_{1k} + C_k &\Leftrightarrow S_{1j} + C_j < S_{1k} + C_k \\ &\Leftrightarrow S_{ij} + C_j < S_{ik} + C_k \\ &\Leftrightarrow C_{i-1} + S_{ij} + C_j < C_{i-1} + S_{ik} + C_k.\end{aligned}$$

If we were to label each matrix element with its rank within a sorted row, the implications above say that within a column all such labels are identical. A similar observation holds if we label elements with their column sorted rank. By sorting the first row we can create an array τ where $\tau(i) = j$ if the ith smallest element of a row is found in the jth column. Likewise, by sorting some column we can create an array ρ, where $\rho(i) = j$ if the ith largest element of a column lies in row j. ρ and τ are created once in $O(m \log m)$ time. Imagine now that we create a *sorted dominance matrix* by physically re-arranging the dominance matrix columns so that the rows are ordered, and physically re-arranging the rows so that the columns are ordered. The sorted matrix has the desired lattice like partial ordering. We can use the same search technique as before on the sorted matrix. It is not necessary though to create the sorted matrix. Whenever we need to access the ij element of the sorted matrix, we create the $\rho(i)\tau(j)$ element of the dominance matrix.

The $O(m \log m)$ cost of creating τ and ρ is masked by the $O(nm \log m)$ cost of calling PROBE2 $O(m)$ times. The overall complexity is again $O(nm \log m)$. Even lower complexities are possible if we employ the linear array itself to solve the mapping problem.

4.3 A Parallel Approach

One approach to parallelizing our serial algorithm is to call the same $O(m)$ probe values as the serial algorithm, using the linear array to compute PROBE2 in parallel. The only opportunity for parallelism here is to parallelize the search over $[k_{\min}, k_{\max}]$, and then combine the individual minimums found by the processors. It takes each processor $O(\log m)$ time to find the interval endpoints, constant time to find a minimum over its

designated subregion of the interval, and then $\Omega(n)$ time to find the global minimum. Asymptotically we lose with this scheme: the complexity of a single PROBE2 call is $O(n \log m + n^2)$.

A different approach is to have each processor perform a set of PROBE2 calls independently, and in parallel with other processors. The strategy we propose is to decompose the implicitly sorted dominance matrix into n regions which are assigned to the processors. Each processor probes its space to find the optimal assignment within that space; an $O(n)$ time combination of results finds the optimal mapping.

We assume that every processor has enough memory to solve the problem alone. The module and communication weights are initially loaded into the processors. Each processor serially computes its own copy of all sums of the form S_{1k}, its own copy of the *right_min* array, and its own copy of τ and ρ. Each processor is now in a position to probe some region of the bottleneck space. The geometry of the regions we choose has an impact on the complexity. An analysis similar to the one presented for the serial case shows that the number of probe calls required to evaluate an $h \times l$ (where $h \leq l$) rectangle is $O(h + h \log(l/h))$. Under the constraint that $h \cdot l$ is constant, it is not difficult to see that we want to make h as small as possible. The optimal approach is to assign each processor a $(m/n) \times m$ region of the sorted dominance matrix. The parallel time complexity is then the sum of an $O(m \log m)$ cost to load the problem and create auxiliary data structures, an $O(m \log m \log n)$ cost to perform the searches in parallel, and an $O(n)$ cost to combine the processor's individual optimal solutions. The $O(m \log m \log n)$ cost dominates.

5 Shared Memory Problem

Our approach to the shared memory problem again uses a probe. We first show how to reduce the cost of a probe based on Kernighan's algorithm [8] from $O(m^2)$ to $O(m \log m)$. We then adopt the same search strategy as we did for the linear array problem and achieve an $O(m^2 \log m)$ time algorithm. Finally, we discuss three approaches for parallelization. One approach divides the sorted dominance matrix into regions which are searched in parallel. This approach yields an algorithm with an $O((m^2/n) \log m \log n)$ time complexity, and $O(nm)$ space complexity. A second approach uses a parallel sort, and then serialized binary search and probe calls. This algorithm reduces the expected time complexity to $O((m^2/n) \log m)$, but increases the space complexity to $O(m^2)$. Our third approach parallelizes the probe function, and is appropriate when $n \ll m$. Under technical conditions on n and m, its expected time complexity is $O((m^2/n) \log m)$, and its space complexity is only $O(m)$.

5.1 An Improved Serial Solution

Iqbal's approximation method cites an algorithm described by Kernighan[8]. The algorithm partitions a chain of modules, subject to the contiguity constraint, and also subject

to the constraint that the sum of module weights in any partition is less than some fixed and pre-determined value w. The cost of a partitioning is the sum of the costs of links exposed by the partitioning. He formulates this problem using dynamic programming, and solves the optimality equations

$$\begin{aligned} V(0) &= 0 \\ V(j) &= C_j + \min_{\substack{i \le j \\ S_{ij} \le w}} \{V(i-1)\} \quad \text{for } j = 1, 2, \ldots m. \end{aligned}$$

$V(j)$ can be interpreted as the minimal cost of partitioning modules M_1 through M_j, including the cost of separating M_j from M_{j+1}. Once $V(m)$ is determined the solution is found by backtracking. If j defines $V(m)$'s min term, then $j+1$ is the left endpoint of the rightmost partition; if i determines $V(j)$'s min term, then $i+1$ is the left endpoint of the next partition, and so on.

This function can be used as a probe. If the chain can be partitioned into n or fewer pieces subject to the partition loading constraint, then the partition defines a feasible mapping; furthermore, it minimizes the sum of communication costs among all mappings with processor loads less than w. The probe compares the sum of communication costs with the probe constraint w; if that sum is smaller, and if n or fewer partition elements are defined, it returns the value "true". So long as w is kept fixed for all problem sizes this solution has $O(m)$ complexity. However, we vary w with every call to the probe function. In the worst case w is W_T, the sum of all module weights, and the algorithm is $O(m^2)$. Iqbal missed this fact, and in [7] ascribes an $O(m)$ complexity to this algorithm.

Kernighan's treatment considers w to be constant, so that the min term for every $V(j)$ can be determined in constant time with a linear scan. Since our w's will vary and may become quite large, we need to avoid linear scans. The min term can be efficiently found with the aid of a search tree which organizes domain points on the basis of their V values. The tree initially contains a single record corresponding to the boundary condition $V(0) = 0$. A pointer *where_is*(0) to that record is stored to aid a future deletion. Subsequently, we compute each $V(j)$ by first identifying the indices over which its min term ranges. The minimal index $i_{\min}$ satisfying $S_{ij} \le w$ can be found with a binary search on $S_{1j}, S_{2j}, \ldots, S_{jj}$, and the *where_is* pointers are used to remove all tree records for $V(i)$ with $i < i_{\min}$. The search tree is then examined for the entry whose key is least; this entry defines $V(j)$'s min term. $V(j)$ is computed by adding the min term and C_j. A record representing $V(j)$ is inserted into the tree, and the pointer *where_is*(j) to that record is saved. The auxiliary value *back_ptr*(j) is set equal to the index of the position defining $V(j)$'s min term.

$O(m)$ tree insertions and deletions costs $O(m \log m)$ amortized time using splay trees[15].

The improved probe function can be used in conjunction with the search strategy described for the linear array problem. Note that a dominance matrix with S_{ij} type entries suffices. Letting $S(b)$ denote the minimized sum of communication costs with b as bottleneck constraint, recall that at the termination of the binary search we will have determined the smallest bottleneck value $\hat{b}$ such that $\hat{b} \ge S(b)$. The optimal sum-bottleneck

solution is then either b or $S(\tilde{b})$, where $\tilde{b}$ is the greatest bottleneck value less than b. Since $\tilde{b}$ may be the solution we seek, it is important to be able to access it quickly. Suppose that throughout the search we maintain a value V_n, the smallest bottleneck value larger than V_s (the least known feasible solution). We claim that $\tilde{b}$ must either be the value of V_n at the end of the search, or be adjacent to b's location in the sorted dominance matrix. The claim is established by contradiction—suppose that $\tilde{b}$ is not V_n and is not adjacent to b. $\tilde{b}$ is eliminated from consideration as the smallest bottleneck exceeding its associated communication cost in one of two ways. $\tilde{b}$ may be eliminated because a smaller bottleneck value satisfies the probe. This bottleneck value can only be b, and would have to be adjacent to $\tilde{b}$, a condition we have assumed does not occur. $\tilde{b}$ can also be eliminated if a larger bottleneck value fails the probe. However, this is impossible because $\tilde{b}$ itself passes the probe. This establishes the contradiction, and thus the fact that given b and V_n, $\tilde{b}$ can be found in constant time.

The cost of $O(m)$ probe calls, each with complexity $O(m \log m)$, is $O(m^2 \log m)$. Note that this same complexity is achieved if we sort the $O(m^2)$ bottleneck values and call the probe $O(\log m)$ times. However, the former approach needs $O(m)$ space, while the latter requires $O(m^2)$ space.

5.2 A Suite of Parallel Approaches

Three different approaches for parallelizing the algorithm suggest themselves. One mimics our parallel linear array solution, and simply divides the dominance matrix into $(m/n) \times m$ sized regions which are searched in parallel. Each region requires $O((m/n) \log n)$ probe calls, a cost which dominates the cost of combining the various processors' optimal solutions. The overall time complexity of this approach is $O((m^2/n) \log m \log n)$. Each processor requires $O(m)$ space.

A second approach is to compute and sort the $O(m^2)$ bottleneck values in parallel. Techniques such as those described in [11], and [19] are appropriate, and have an $O((m^2/n) \log m)$ expected parallel complexity. A binary search over the sorted values may then be employed, with a serial probe. $O(\log m)$ probe calls are made, each with $O(m \log m)$ complexity. The resulting algorithm has an $O(\max\{(m^2/n) \log m, m \log^2 m\})$ expected parallel time complexity, but requires $O(m^2)$ space for the sort.

An $O((m^2/n) \log m)$ expected time complexity with $O(m)$ space requirements is possible in the event that $8n^3 < m$. In this case we can effectively parallelize the probe function. Our approach relies on the likelihood that if $V(i)$ defines the min term for $V(j)$, then $i \ll j$. If $V(j)$ does not depend on "nearby" values of V, then "nearby" values of V can be computed in parallel. Of course, if $V(i)$ and $V(j)$ are computed in parallel and it turns out that $V(j)$'s min term should have been $V(i)$, then we need to recompute $V(j)$. We will see though that this occurs infrequently under our stochastic assumptions about module and weight values. It should be noted that unlike the other complexities derived in this paper, the magnitudes of the constants of proportionality are not obviously low. Without

further discussion on this topic, we note here that when the module weight distribution's *coefficient of variation* σ/μ is low, then the constants of proportionality are low.

A general description of the algorithm follows. We divide the domain into successive blocks $B_1, B_2, \ldots, B_{m/n}$, of n consecutive points each. We will compute all values of V within a block in parallel, assigning one processor per block point. The processors create and combine information describing the solution of V in the block area, and check to ensure that no value computed in the block depends directly on another value within the same block. If such a dependency is detected it can be corrected with a serialized computation of the block values. Once the block values are correct the processors move on to the next block. The backtracking phase to find the optimal partition is serial. We turn next to a more detailed description of this procedure.

The algorithm begins with every processor initializing its own search tree such as was used in the serial version. The search tree may reside in the processor's local memory. The global memory will contain the V array. Processor P_i then computes $V(i)$. Since $n \ll m$, it is unlikely that the probe weight w will be small enough so that $S_{1n} > w$, and it is highly likely that $V(i) = C_i$ is the correct value for $V(i)$. The processors cooperatively compute the minimum value $m_1 = \min_{1 \le i \le n}\{V(i)\}$. It is well-known that this can be done in $\log n$ steps with a combining tree as shown in figure 8(a). The entire tree is left in the global memory. Note however that communication is serialized, implying that the cost of building the tree is $O(n)$. Figure 8(b) illustrates the fact that the minimum value of V over the last k items of a block can always be recovered from the combining tree by examining no more than $\log n$ entries. If $S_{1n} \le w$ then m_1 is the minimum value of V over the first block. Every processor inserts m_1 into its local search tree, and for the purposes of future deletion records a pointer to its location.

The computation now proceeds in stages. The values for B_k are computed by the kth stage with the following operations.

1. *Serial Step:* Note that B_k consists of integers in $[(k-1)n+1, kn]$. We must first determine whether it is feasible to compute all of B_k's points in parallel. A necessary condition for this is that the indices of $V(kn)$'s min term completely encompass B_k. This is checked by determining whether $S_{((k-1)n)(kn)} \le w$. If not, then we cannot evaluate all of B_k's points in parallel. In this case we serialize the computation of the block, and advance to the next block.

2. *Parallel Step:* Processor P_j is responsible for computing $V((k-1)n+j)$. P_j first uses a binary search to find the left endpoint $i_{\min}(j)$ of the indices over which its min term is taken. P_j then deletes from its search tree all entries representing blocks including and lying to the left of $i_{\min}(j)$. Let $i_r(j)$ be the right endpoint of the block containing $i_{\min}(j)$, and let $v_l(j)$ be the minimum value of V over $[i_{\min}(j), i_r(j)]$. $v_l(j)$ can be found by examining the combining tree over $i_{\min}(j)$'s block.

3. *Parallel Step :* Processor P_j finds the minimum value $v_s(j)$ within its own search

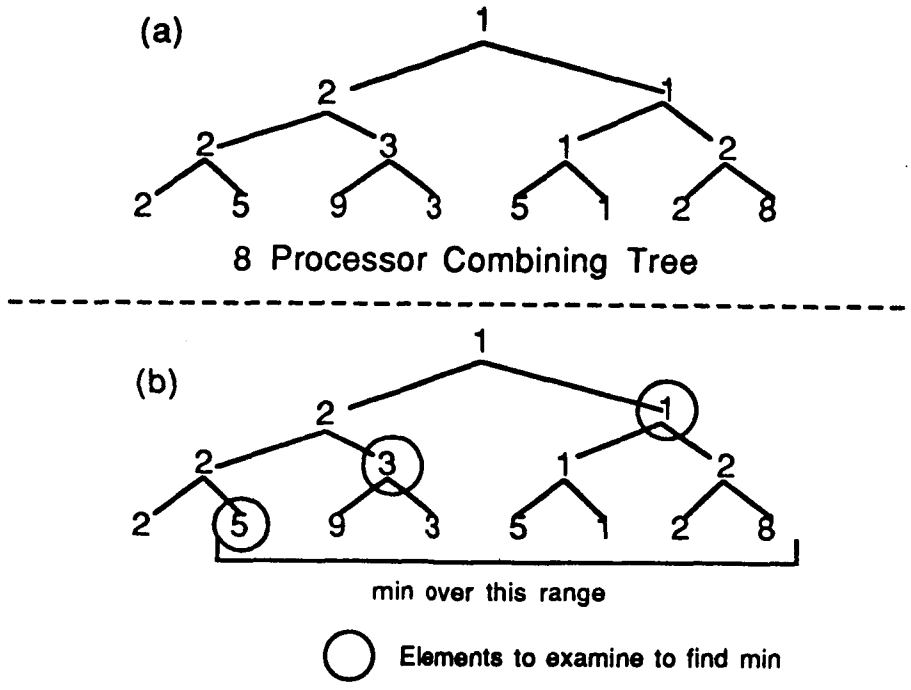

Figure 8: Combining tree to compute the minimum of n values

tree. Then P_j computes $V((k-1)n+j) = C_{(k-1)n+j} + \min\{v_l(j), v_s(j)\}$, and records in local memory a *back_ptr* value giving the index which defines $\min\{v_l(j), v_s(j)\}$.

4. *Parallel Step:* The processors cooperatively compute the minimum value v_b of V over the current block, with a combining tree.

5. *Serial Step:* P_n checks to see if its current V value is correct, by comparing $V(kn)$ with $C_{kn} + v_b$. If the latter quantity is smaller, then the earlier computation was incorrect. Because the range of $V(kn)$'s min term includes all of B_k, if any V computed in B_k is incorrect, $V(kn)$ will be incorrect and will be detected. When this occurs, the block's points are recomputed serially.

Over the course of the algorithm, an individual processor inserts, deletes, and searches for m/n items in the search tree. Collectively this exacts an $O((m/n)\log(m/n))$ amortized time cost. In the absence of serialization, for each of m/n stages, step (1) takes $O(1)$ time; noting that communication is serialized, step (2) takes $O(\max\{n\log n, \log m\})$ time; step (3) takes $O(\log(m/n))$ time; step (4) takes $O(n)$ time due to serialized communication, and step (5) takes $O(1)$ time. In the absence of serialization the overall complexity depends on the relationship between m and n. If $n\log n > \log m$, then the $O(n\log n)$ cost of step (2) dominates and the algorithm has an $O(m\log n)$ cost. If $n\log n \leq \log m$, then the $O(\log m)$

cost of step (2) dominates, yielding an $O((m/n)\log m)$ algorithm. As m grows we expect that eventually the latter case will hold; for simplicity in exposition we assume that m is sufficiently larger than n to give an $O((m/n)\log m)$ parallel time complexity in the absence of serialization.

If the computation is serialized, a shared variable can indicate which processor is allowed to compute its value. A processor proceeds as before, except that the minimum value of V seen so far within the block must also be considered in step (3). Each point calculation takes $O(\log m)$ time, so the entire block takes $O(n \log m)$ time.

Without serialization the parallel complexity of this probe is $O((m/n)\log m)$. Serialization may occur at step (1) when w is too small in relation to n. Because the m modules must be distributed over only n processors, we expect that each processor receives on the order of m/n modules, and that the values passed to the probe tend to be from convolutions of approximately m/n module sums. Intuitively then we see that serialization shouldn't occur often, provided that m is sufficiently larger than n. The subsection to follow shows that if $8n^3 < m$ then serialization occurs so infrequently that the expected complexity of the entire algorithm is $O((m^2/n)\log m)$.

5.3 Expected Complexity When $8n^3 < m$

If we can reduce the frequency of serialization to $O(1/n)$, the contribution of serialization to the algorithm's overall complexity will be $O((m^2/n)\log m)$ which is exactly the parallel complexity. We will show that this occurs when m is sufficiently larger than n. We do so in three steps. First we show that if $w > 2n\mu_m$, then the probability of serialization being required at step (1) of the parallel probe is $O(1/n)$. Secondly, we show that if $w > n^2\mu_m/2$, then the probability of serialization being required at step (5) of the parallel probe is also $O(1/n)$. Finally, under some simplifying assumptions we show that when $8n^3 < m$, then probe calls with w values less than $n^2\mu_m/2$ occur so infrequently that the expected cost due to serialization is only $O((m^2/n)\log m)$.

Consider the parallel probe function. The first chance at serialization occurs in step (1). Let $p_1(w)$ be the probability that the sum of n module weights associated with a block exceeds w. We assume that every module execution time is drawn independently from a common distribution with finite mean μ_m and standard deviation σ_w. Likewise, we assume that the communication costs are independent and identically distributed, although they are allowed to be from a different distribution. Our analysis rests on two facts from probability theory.

- If $X_1, X_2, \ldots, X_k$ are k independent identically distributed random variables with mean μ and standard deviation σ, then the mean of the linear combination $\sum_{i=1}^{k} a_i X_i$ is $\mu \sum_{i=1}^{k} a_i$, and the standard deviation is $\sigma\sqrt{\sum_{i=1}^{k} a_i^2}$.
- *Chebychev's Inequality* If X is any random variable with mean μ and standard

deviation σ, and ϵ is any positive number, then

$$Prob\{|X - \mu| > \epsilon\sigma\} \leq \frac{1}{\epsilon^2}.$$

These facts may be found in any standard probability text, such as[10].

Let $M(n)$ be an n-fold convolution of the module weight distribution. $M(n)$ has mean $n\mu_m$ and standard deviation $\sigma_m\sqrt{n}$. Serialization is chosen at step (1) if the sum of the block's n module weights exceeds w. Appealing to a slightly re-organized form of Chebychev's inequality we have

$$Prob\{M(n) > n\mu_m + \epsilon\sigma_m\sqrt{n}\} \leq \frac{1}{\epsilon^2}.$$

for any positive constant ϵ. Choosing $w = n\mu_m + \epsilon\sigma_m\sqrt{n}$ and solving for ϵ, we have

$$\begin{aligned} p_1(w) &= Prob\{M(n) > w\} \\ &= Prob\{M(n) > n\mu_m + \epsilon\sigma_m\sqrt{n}\} \\ &\leq \frac{n\sigma_m}{(w - n\mu_m)^2} \end{aligned}$$

whenever $w > n\mu_m$. If $w > 2n\mu_m$, then the right hand side of this inequality is $O(1/n)$. We have proved the following theorem.

Theorem 1 *Let $p_1(w)$ be the probability of serialization at step (1). If $w \geq 2n\mu_m$, then $p_1(w) = O(1/n)$.*

Now let $p_2(w)$ be the probability that serialization is chosen in step (4). This occurs when the min term of some $V(j)$ is defined by some V value in $V(j)$'s block. To show that $p_2(w) = O(1/n)$ when $w > n^2\mu_m/2$ we will need the following technical lemma.

Lemma 2 *For every $j = 1, 2, \ldots, m$ let*

$$L(j, w) = \{V(i) \mid i < j, S_{ij} \leq w\}.$$

Then for all j and w, $\min L(j, w) \geq \min L(j-1, w)$.

Proof Suppose $L(j-1, w) = \{V(i_l), \ldots, V(j-2)\}$ and $L(j, w) = \{V(i_u), \ldots, V(j-1)\}$. Note that i_l is necessarily no greater than i_u. This implies that

$$\min\{V(i_u), \ldots, V(j-2)\} \geq \min L(j-1, w).$$

Now

$$L(j, w) = \{V(i_u), \ldots, V(j-2)\} \cup \{V(j-1)\}$$

so that

$$\begin{aligned}\min L(j,w) &= \min(\{V(i_u),\ldots,V(j-2)\} \cup \{C_{j-1} + \min L(j-1,w)\}) \\ &\geq \min(\{V(i_u),\ldots,V(j-2)\} \cup \{\min L(j-1,w)\}) \\ &= \min L(j-1,w).\end{aligned}$$

□

The main purpose of lemma 2 is to aid in the proof of the following lemma.

Lemma 3 *Let $V(i), V(i+1), \ldots, V(i+N-1)$ be a consecutive sequence of V values. Then the probability that the minimum value occurs in one of the last n sequence elements is no greater than n/N.*

Proof Let q_j be the probability that $V(i+j)$ is the minimum in the sequence. We first show that

$$q_0 \geq q_1 \geq \cdots \geq q_{N-1}.$$

Consider the module weights to be fixed, but let the communication weights be random. Let $J =< c_i, c_{i+1}, \ldots, c_{i+N-1} >$ be any random vector sampled from the joint distribution of the communication costs, and suppose that under this joint vector $V(i+k)$ is minimum. By lemma 2, $\min L(i+k-j,w) \leq \min L(i+k,w)$ for all j such that $1 \leq j \leq k$. Since $V(i+k-j) = \min L(i+k-j,w) + c_{i+k-j} > \min L(i+k,w) + c_{i+k} = V(i+k)$, we must have $c_{i+k-j} > c_{i+k}$. Suppose we swapped the costs c_{i+k} and c_{i+k-1}. The swap does not affect any $V(i+k-j)$ with $j > 1$, but clearly $V(i+k-1) < V(i+k)$. Furthermore, any V value to the left of $V(i+k-1)$ is larger, because

$$\begin{aligned}\min L(i+k-j,w) + c_{i+k-j} \;>\; &\min L(i+k,w) + c_{i+k} \quad \Rightarrow \\ &\min L(i+k-j,w) + c_{i+k-j} > \min L(i+k-1,w) + c_{i+k}.\end{aligned}$$

Any value to the right of $V(i+k-1)$ must also be larger—the *min term* for some values $V(i+k+j)$ to the right of $V(i+k-1)$ may change to either the new value of $V(i+k)$ or $V(i+k-1)$, but the *new value* of $V(i+k+j)$ cannot be less than the new value of $V(i+k-1)$. Because the communication costs are independent and identically distributed, the random vector which swaps the values c_{i+k-1} and c_{i+k} in J has the same probability mass or density as J. Consequently, for any random sample where $V(i+k)$ is minimum there is an equally likely sample where $V(i+k-1)$ is minimum. As this is true for any sampling of module execution weights, we must have

$$q_0 \geq q_1 \geq \cdots \geq q_{N-1}.$$

For any descending sequence of N values, it is always true that the sum of the last n elements is no greater than n times the sequence average. The sequence average here is

$1/N$ because the q_j's must sum to 1. The probability that the minimum occurs in one of the last n positions is the sum of the last n sequence values, and consequently is no greater than n/N.

□

At step (5) serialization is required at block B_k if $V(kn)$'s min term is defined by some value in B_k. Lemma 3 tells us that if $L(kn, w)$ has N elements, then the probability of serialization is no greater than n/N. If we can keep the size of $L(kn, w)$ on the order of n^2, then serialization occurs at step (5) with $O(1/n)$ probability. The size of $L(kn, w)$ is a random variable which we call $N^*(w)$. $p_2(w)$ is no greater than the expected value of $nE[1/N^*(w)]$. The theorem to follow bounds this expectation by $O(1/n)$ in the event that $w > n^2\mu_m/2$.

Theorem 4 *If $w > n^2\mu_m/2$, then $p_2(w) = O(1/n)$.*

Proof

$$\begin{aligned} p_2(w) &= Prob\{\text{one of } B_k's \ V \text{ terms is minimum in } L(kn, w)\} \\ &\leq nE[1/N^*(w)] \end{aligned} \tag{1}$$

where the expectation is taken with respect to the distribution of $N^*(w)$. The function $f(x) = 1/x$ is decreasing, and is bounded from above by $g(x)$, defined below:

$$g(x) = \begin{cases} 1 & \text{If } 1 \leq x < n^2/4 \\ 4/n^2 & \text{If } x \geq n^2/4 \end{cases}.$$

Because $g(x) \geq f(x)$ for all x, we must have $E[g(N^*(w)] \geq E[1/N^*(w)]$. Now $N^*(w)$ is less than $n^2/4$ only if the sum of $n^2/4$ or fewer module weight random variables is greater than $n^2\mu_m/2$. The proof of Lemma 1 bounded a very similar probability using Chebychev's inequality. Applying the same methodology here, it can be shown that the probability of $N^*(w)$ being less than $n^2/4$ is $O(1/n^2)$, if $w > n^2\mu_m/2$. We then have

$$\begin{aligned} E[g(N^*(w))] &= Prob\{M(n^2/4) > w\} \cdot 1 + Prob\{M(n^2/4) \leq w\} \cdot \frac{4}{n^2} \\ &= O(1/n^2) \end{aligned}$$

Applying this to relation (1), the lemma's conclusion follows.

□

Theorems 1 and 4 tell us that if the probe weight w is large enough then serialization occurs infrequently. We next show that if m is sufficiently larger than n we can expect the probe weights used by our algorithm to be large enough to satisfy the theorems' conditions. A note of warning is in order. The results to follow relate to pristine convolutions of the module weight distributions. The values of w chosen by our search procedure are indeed sums of module weights, but the distribution of those sums are affected by the history of

the search behavior. For example, suppose we choose to probe with value S_{ij}, found in the upper left rectangle identified by the the first rectangle evaluation. S_{ij} is not identically distributed with a sum of $j-i+1$ independent module weights. We know that the probe is satisfied on S_{ik} for some $k > j$—this was established by the first rectangle evaluation. We also know that for some $k > i$, S_{kj} fails the probe. The former observation tends to make S_{ij} "larger" probabilistically, because some portion of the chain it represents is involved with sums known to succeed. Likewise, the latter observation tends to make S_{ij} "smaller" because the M_k to M_j subchain weight must fail the probe. The affects of the search behavior on probe value distributions appear to be too complex to deal with analytically. But because of the conflicting influences on the probe value distribution it seems likely that these effects on the size of the probe values are second order compared to the effects on pure module weight convolutions of increasing the size of the sums. By assuming that the probe weights are drawn from pure module weight convolutions, we *can* make statements about the probability of the probe function being satisfied.

The discussion to follow speaks in terms of w being drawn from a convolution of k module weights, where k may vary. We have already used $M(k)$ to denote a k-fold convolution of module weights. To say that w is drawn from that convolution we will write $w \sim M(k)$. For our purposes three bounds are quite important and are summarized by the following lemma.

Lemma 5 *Let $M(k)$ denote a k-fold convolution of module weight random variables. Then*

(a) $Prob\{M(k) > 2k\mu_m\} = O(1/k)$.

(b) $Prob\{M(k) < k\mu_m/2\} = O(1/k)$.

(c) *If $M_1(k)$ and $M_2(2k)$ are independent convolutions, then* $Prob\{M_1(k) > M_2(2k)\} = O(1/k)$.

Proof (a) and (b) are found in a manner entirely similar to the proof of Theorem 1. (c) is found in the same fashion by first noting that

$$Prob\{M_1(k) > M_2(2k)\} = Prob\{M_1(k) - M_2(2k) > 0\},$$

and that the random difference has mean $-k\mu_m$ and standard deviation $\sigma_m\sqrt{3k/2}$.
□

An important component of our search strategy is to call the probe function with bottleneck value w only if w exceeds V_f—the greatest probe value known to fail, and if w is dominated by V_s—the best known solution to date. This test offers protection from serialized probe calculations when the probe value w touched by the search is small; with high probability $w < V_f$. Let I_f be the number of modules summed to form the value of V_f immediately after the first rectangle search. We will say that the search is *irregular* if $I_f < m/4n$ or I_f is undefined, and otherwise is *regular*. For the purposes of bounding

costs we will assume that any irregular search is completely serial, but then show that the probability of an irregular search is so low that the expected cost due to irregular searches is $O((m^2/n^2)\log m)$. We accomplish this by showing that the probability of an irregular search is $O(1/n^2)$.

Suppose that I_f is defined, and equals $k < m/4n$. This implies that some w drawn from a convolution of $k+1$ modules actually satisfies the probe. For simplicity we assume that $w \sim M(k+1)$, although this is not rigorously true. The probability that a $M(k+1)$ random variable satisfies the probe is no greater than the probability that $w \sim M(m/4n)$ satisfies a *new probe* which passes automatically if $w > m\mu_m/2n$, and which calls the original probe otherwise. The new probe is constructed only for the purpose of bounding probabilities. The probability of the new probe passing automatically is the probability that $w > m\mu_m/2n$; but since $w \sim M(m/4n)$, lemma 5(a) says this probability is $O(4n/m)$. As $8n^3 \leq m$, the probability of the new probe passing automatically is $O(1/n^2)$. The new probe is also satisfied if $w < m\mu_m/2n$, and the old probe passes w. A necessary condition for the old probe to pass w is that each of n processors receives a load less than or equal to w. This implies that the sum of all module weights can be no greater than nw. Given that $w < m\mu_m/2n$, the sum of all module weights can be no greater than $m\mu_m/2$. But by lemma 5(b) the probability of this occurring is $O(1/m)$. Finally we consider the possibility that I_f is not defined. For this to occur the least weight on the first strip must pass the probe, a weight composed of a single module weight. The same types of arguments as used above will obviously establish that the chance of this occurrence is infinitisimal. Consequently, the chance of an irregular search is $O(1/n^2)$.

Now we show that the expected complexity of a regular search is $O((m^2/n)\log m)$. Since the search is regular we have $I_f \geq m/4n$. Let w be a weight touched by the search. Two cases may occur.

Case 1 Suppose that w is composed of k module weights, and $k < m/8n$. For simplicity we assume that $w \sim M(k)$. A necessary condition for actually calling the probe function is that $M(k)$ exceed the value $\hat{V}_f$, the value of V_f immediately after the first rectangle evaluation. w is not independent of $\hat{V}_f$, but we will assume so for the sake of tractability. The probability of calling the probe function is then bounded by the probability that a convolution $M_1(m/8n)$ exceeds another independent convolution $M_2(m/4n)$. By lemma 5 the probability of this occuring is $O(n/m) = O(1/n^2)$. If we assume that an actual probe call must serialize because w is too small, then the expected cost due to this occurrence is only $O((m^2/n^2)\log m)$.

Case 2 Suppose that w is composed of k module weights, and $k \geq m/8n$. For the purposes of bounding costs, suppose that if $w < m\mu_m/16n$ then the search serializes. By lemma 5(b) the probability of this is $O(n/m) = O(1/n^2)$, and the expected cost of serialization in this fashion is $O((m^2/n^2)\log m)$. But if $w \geq m\mu_m/16n$, then $w > n^2\mu/2$ because we have assumed $8n^3 \leq m$. By theorems 1 and 4 the probability of serialization is only $O(1/n)$.

Finally, we must consider the behavior of the search during the first rectangle evaluation. While unlikely, the worst case occurs if each of $\log m$ probes serializes. The cost of evaluating the first rectangle is then $O(m \log^2 m)$. However, when m is sufficiently larger than n ($m^{2/3} > \log m$) this cost is dominated by the parallel $O((m^2/n) \log m)$ complexity.

The discussions above have shown that when $8n^3 \leq m$ then the overall expected time cost due to serialization is $O((m^2/n) \log m)$. The expected cost in the absence of serialization was also $O((m^2/n) \log m)$, making this expression the overall expected time complexity The space required for the parallel probe is only $O(m)$.

6 Host-Satellite Problem

Our approach to the host-satellite problem is again modeled on Iqbal's probing approach. For a given bottleneck value w we apply a PROBE2-like function (from the linear array problem) to each satellite chain. The bottleneck weights are all of the form Ω_{1j}, where the Ω function is identical to that of the linear array problem. This probe will load the satellite with the feasible load which minimizes the Δ function. The unassigned load is given to the host, and the communication cost of breaking the chain is suffered by both the host and the satellite. The host's cost is the sum of the n off-loaded subchains, the associated communication costs, plus some additional load H which it must always compute. Since each satellite minimized the load given to the host under the bottleneck contraint on satellite loads, the host's load is minimized. The probe returns true if the host's load is no greater than the bottleneck weight. As before, we will first improve upon the known serial solutions, and then show how to parallelize the mapping algorithm. We will reduce the serial time complexity to $O(\max\{nm \log m, n \log^2 m\})$, and find a parallel solution with $O(\max\{nm, n \log m \max\{n, \log m\}\})$ complexity. When m is sufficiently larger than n the nm term will dominate; in this case the complexity is within a constant factor of optimal under the assumption that $O(nm)$ time is required to load the problem onto host-satellite system.

6.1 An Improved Probing Approach

The set of bottleneck weights for the host-satellite problem has a different structure than that of the previous two problems, but it is still exploitable. The bottleneck weights for a given chain are of the form $C_0 + S_{1j} + C_j = \Omega_{1j}$, and consequently are not necessarily monotone increasing in j. It is important to remember that each chain has its own set of Ω values. To allow the possibility of moving a satellite's entire chain onto the host we define $\Omega_{10} = C_0$, where C_0 is the communication cost of transmitting the satellite's incoming data to the host. The assumption that communication costs are bounded allows us to sort a chain's bottleneck values in $O(m)$ time, using brute force. Define arrays *right_less* and *left_greater*, each with $m + 1$ entries, and all entries initialized to zero. At the end of the algorithm *left_greater*(j) will contain the number of bottleneck values Ω_{1i} such that

$i < j$, and $\Omega_{1i} > \Omega_{1j}$. Similarly, *right_less*(j) will contain the number of bottleneck values Ω_{1k} such that $k > j$, and $\Omega_{1k} < \Omega_{1j}$. Ω_{ij}'s rank (rank 0 meaning smallest) in the sorted list is consequently *right_less*$(j) + j -$ *left_greater*(j). The trick is to efficiently compute the auxiliary arrays. For every $j = 0, \ldots, m$ we scan increasing values of Ω_{1k}, $k > j$ incrementing *right_less*(j) and *left_greater*(k) every time we encounter a k such that $\Omega_{1k} < \Omega_{1j}$. The important point is that we may stop scanning as soon as k is so large that $C_j < S_{jk}$, because we are assured that Ω_{1k} for larger k is always larger than Ω_{1j}. Because the communication costs are bounded, these scans require constant time. Given the ranks, the items can be sorted in $O(m)$ time. This gives the sorting algorithm an $O(m)$ complexity.

$O(nm)$ time is required to compute the auxiliary data structures for the probe function, and to sort each of n vectors of bottleneck values. The n sorted vectors can be merged into a single sorted list in $O(nm \log n)$ time. A binary search over the sorted list of bottleneck values with a probe call at each touch has $O(n \log^2 m)$ complexity. As before, we must also consider the next smallest bottleneck weight $\tilde{b}$ which passes the probe. $\tilde{b}$ must lie adjacent to the bottleneck value found by the search and so is considered in constant time. Depending on the relationship between n and m, the overall complexity is either $O(nm \log n)$ or $O(n \log^2 m)$; in either case an improvement over Bokhari's $O(nm^2 \log m)$ solution, or our $O(nm \log m)$ improvement upon Bokhari's solution.

6.2 A Parallel Approach

The sorting step dominates the complexity of our serial algorithm. If we treat the host like a shared memory, then the satellites could conceivably sort the bottleneck values in parallel. However, in all likelihood a real host-satellite system will not emulate a shared-memory machine particularly efficiently, so that we should practically consider another approach.

An easy way to exploit parallelism is to perform the probe function in parallel. The natural way to do this is to have each satellite call a PROBE2-like function on its own subchain structure. To support such an approach, each satellite is loaded with its own subchain costs. In parallel, each satellite sorts its own Ω values as previously described. The probe values will be selected by performing a binary search over each satellite's list of bottleneck weights; first we search the entire list of the first satellite, then the entire list of the second satellite, and so on. For every probe touch the host can query the appropriate satellite for the proper probe value, and then transmit that value to every satellite. Each satellite then calls a PROBE2-like function to determine the feasible load which minimizes the remaining load (which is the host's cost), and reports the remaining load to the host. The host computes its own load and determines whether the probe passed or failed. Loading the problem onto the satellites takes $O(nm)$ time. Each parallel probe call takes $O(\max\{n, \log m\})$ time; there are $O(n \log m)$ probe calls. The overall parallel time complexity is $O(\max\{nm, n \log m \max\{n, \log m\}\})$. When m is sufficiently

larger than n the $O(nm)$ cost of loading the problem dominates. In this case the algorithm is within a constant factor of optimal, if we assume that the time to load the problem onto the host-satellite system is proportional to the problem size.

7 Summary

We have examined three parallel mapping problems: mapping a chain of modules onto a linear array, a chain of modules onto a shared memory machine, and mapping a set of chains onto a host-satellite system. In each case we determine the mapping which minimizes the computation's finishing time, subject to a contiguity constraint. These problems were originally shown to be tractable by Bokhari in [4]. Our work builds on his by first showing that his solutions can immediately be improved by a factor of m (the number of modules), and then by demonstrating that there are much more efficient solutions than those that demonstrated the problems' tractability. In addition, we showed how the target parallel architectures themselves can be used to compute the optimal mapping. In some cases we showed that algorithms with bad worst case complexity have good average case complexity. The table below compares the time complexities of Bokhari's original algorithms, our improvement on those algorithms, Iqbal's approximation methods, our serial and parallel improved methods. In some cases we have simplified complexities by assuming that m is much larger than n.

Problem	**Bokhari**	**Improved Bokhari**	**Iqbal** *(Approximate)*	**Improved Serial**	**Parallel**
Linear Array	nm^3	nm^2	$mn \log(W_T/\epsilon)$	$nm \log m$	$m \log m \log n$
Shared Memory	$nm^3 \log m$	$nm^2 \log m$	$m^2 \log(W_T/\epsilon)$	$m^2 \log m$ *(amortized)*	$(m^2/n) \log m$ *(expected, amortized)*
Host-Satellite	$nm^2 \log m$	$nm \log m$	$nm \log(W_T/\epsilon)$	$nm \log n$	nm

Acknowledgements This research would not have been done if Dave O'Hallaron hadn't insisted that the layered graph approach could be expressed in dynamic programming equations. This insistence ultimately led to the improvements in the layered graph algorithms, and the search for better methods. David Middleton acted admirably as a tailor's dummy, and Shahid Bokhari civilly encouraged this work.

References

[1] ANNARATONE, M., ARNOULD, E., GROSS, T., KUNG, H., LAM, M., MENZILCIOGLU, O., AND WEBB, J. The Warp computer: architecture, implementation, and performance. *IEEE Trans. on Computers C-36*, 12 (December 1987), 1523–1538.

[2] BERGER, M., AND BOKHARI, S. H. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers C-36*, 5 (May 1987), 570–580.

[3] BOKHARI, S. *Assignment Problems in Parallel and Distributed Computing.* Kluwer Academic Publishers, Boston, 1987.

[4] BOKHARI, S. H. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers 37*, 1 (January 1988), 48–57.

[5] BOKHARI, S. H. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans. on Soft. Eng. SE-7*, 6 (November 1981), 583–589.

[6] CVENTANOVIC, Z. The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Trans. on Computers C-36*, 4 (April 1987), 421–432.

[7] IQBAL, M. *Approximate Algorithms for Partitioning and Assignment Problems.* Tech. Rep. 86-40, ICASE, June 1986.

[8] KERNIGHAN, G. Optimal sequential partitions of graphs. *Journal of the ACM 18*, 1 (January 1971), 34–40.

[9] KNUTH, D. *The Art of Computer Programming, vol. 1.* Addison-Wesley, New York, 1968.

[10] LARSON, H., AND SHUBERT, B. *Probabilistic Models in Engineering Sciences.* Vol. 1, Wiley, New York, 1979.

[11] NOGA, M. Sorting in parallel by double distributed partitioning. *BIT 27*, 3 (1987), 340–348.

[12] REED, D. A., ADAMS, L. M., AND PATRICK, M. L. Stencils and problem partitionings: their influence on the performance of multiple processor systems. *IEEE Trans. on Computers C-36*, 7 (July 1987), 845–858.

[13] SADAYAPPAN, P., AND ERCAL, F. Nearest-neighbor mappings of finite element graphs onto processor meshes. *IEEE Trans. on Computers C-36*, 12 (December 1987), 1408 1424.

[14] SALTZ, J., NAIK, V. K., AND NICOL, D. Reduction of the effects of the communication delays in scientific algorithms on message passing mimd architectures. *SIAM J. Sci. Stat. Comput 8*, 1 (1987), s118–s134.

[15] SLEATOR, D., AND TARJAN, R. Self-adjusting binary search trees. *Journal of the ACM 32*, 3 (July 1985), 652–686.

[16] STONE, H. Critical load factors in distributed computer systems. *IEEE Trans. on Soft. Eng. SE-4*, 3 (May 1978), 254–258.

[17] STONE, H. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. on Soft. Eng. SE-3*, 1 (January 1977), 85–93.

[18] TOWSLEY, D. Allocating programs containing branches and loops within a multiple processor system. *IEEE Trans. on Soft. Eng. SE-12*, 10 (October 1986), 1018–1024.

[19] YANG, M., HUANG, J., AND CHOW, Y. Optimal parallel sorting scheme by order statistics. *SIAM Journal on Computing 16*, 6 (December 1987), 990–1003.

NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181655 ICASE Report No. 88-2		

4. Title and Subtitle	5. Report Date
PARALLEL ALGORITHMS FOR MAPPING PIPELINED AND PARALLEL COMPUTATIONS	April 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
David M. Nicol	88-2
	10. Work Unit No. 505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18107
	13. Type of Report and Period Covered: Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

Submitted to IEEE Trans. Comput.

16. Abstract

Many computational problems in image processing, signal processing, and scientific computing are naturally structured for either pipelined or parallel computation. When mapping such problems onto a parallel architecture, it is often necessary to aggregate an obvious problem decomposition. Even in this context the general mapping problem is known to be computationally intractable, but recent advances have been made in identifying classes of problems and architectures for which optimal solutions can be found in polynomial time. Among these, the mapping of pipelined or parallel computations onto linear array, shared memory, and host-sattelite systems figures prominently. This paper extends that work first by showing how to improve existing serial mapping algorithms. Our improvements have significantly lower time and space complexities: in one case we reduce a published $O(nm^3)$ time algorithm for mapping m modules onto n processors to an $O(nm\log m)$ time complexity, and reduce its space requirements from $O(nm^2)$ to $O(m)$. We then reduce run-time complexity further with parallel mapping algorithms based on these improvements that run on the architectures for which they are creating mappings.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
parallel algorithms, mapping, parallel processing, pipelines	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	34	A03