Improved Algorithms for Partitioning Tree and Linear Task Graphs on Shared Memory Architecture

Sibabrata Ray Hong Jiang
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE 68588-0115

Abstract

In parallel and distributed computing the overall system performance is significantly influenced by how a task graph representing an application is mapped onto a specific multiprocessor topology. In this paper algorithms with improved performance are proposed to map tree and linear task graphs onto shared memory multiprocessing architecture. Specifically, the task graphs are partitioned with our algorithms so that the load is balanced, processor utilisation is maximized, and the communication demand on the interconnection network is minimized in a shared memory multiprocessor. All algorithms proposed here are polynomial in time complexity. While bottleneck and processor minimization algorithms of the type are proposed for the first time, to the best of our knowledge, the bandwidth minimization algorithm has a complexity of $O(n + p \log q)$ $(q \leq p \leq n)$, in contrast with the complexity of $O(n \log n)$ of the best known algorithm in the literature. Further, we have identified cases where our algorithm will run in linear time on the average.

1 Introduction

The thrust in research in the area of parallel and distributed processing has in general been focussed on improving the performance of parallel applications on multiprocessors either by minimizing turn-around time or by maximizing throughput. One area of particular interest is how to map a given problem onto a multiprocessor of a specific topology and architecture so that a desired, if not optimal, performance can be obtained. This necessitates constructing a partition of the problem into subproblems that achieves the following two goals: 1) the partition is load-balanced and 2) the inter-process (inter-subproblems) communication or the total communication demand on the interconnection network is minimized. Obviously, achieving one goal does not necessarily mean that the other goal will be automatically achieved. In fact, these two goals are often contradictory. It is this contradictory nature of the problem that makes it challenging. The general problem of optimal mapping a problem onto a multiprocessor topology is known to be NP-complete [7].

In general, the structural feature of a multiprocessor can be represented by a graph isomorphic to its topological structure. This *architecture graph* is denoted by $G_{arch} = (P, L)$, where $P = \{p_1, p_2, ..., p_k\}$ is the set of processors and $L = \{l_i \mid l_i = (p_j, p_k) \in P \times P$ where processors p_j and p_k are connected by a network channel $\}$ is the set of network links, though the links need not be independent. Similarly, the structural feature of a parallel application can be represented by a *task graph* $G_{task} = (N, MD)$ where $N = \{t_1, t_2, ..., t_n\}$ is a set of tasks comprising of the application and $MD = \{m_i \mid m_i = (t_j, t_k) \in N \times N$; where tasks t_j and t_k need to exchange messages directly$\}$ is the set of data dependencies. The behavioral properties of a multiprocessor or a parallel application can be described by associating certain *weights* with respective graphs. For instance, a weight, $w(p_i)$, associated with a node p_i in G_{arch} indicates the processing speed of processor p_i (e.g., instructions per second), whereas a weight, $w(l_i)$, on an edge l_i of G_{arch} indicates the communication bandwidth of that network channel (i.e., bits per second). By the same token, the amount of processing requirement, $w(t_i)$ (in number of instructions), is associated with each node t_i of G_{task} and the amount of messages, $w(m_i)$ (in bits), is associated with each edge m_i of G_{task}.

Thus, the general optimization problem can be formulated using the above notation as follows. To achieve the goal of load-balancing, a partition G_{task}, denoted by $\mathbf{P} = \{G_{task_1} = (N_1, MD_1), ..., G_{task_k} = (N_k, MD_k)\}$, to be constructed such that 1) N_i's are disjoint; 2) $\sum_{t_j \in N_i} w(t_j)$, $1 \leq i \leq k$, is approximately $\sum_{t_j \in N} w(t_j)/k = K$ and 3) $\sum_{m_j \in MD} w(m_j) - \sum_{i=1}^{k} \sum_{m_j \in MD_i} w(m_j)$ is minimized. For processor allocation, on the other hand, a mapping $\mathbf{M}$ of $\mathbf{P}$ onto G_{arch} is to be computed such that the processing time on each processor is approximately same and the communication overhead is minimum. Clearly, if the multiprocessor is homogeneous, that is, all processors have the same speed and all network channels have the same bandwidth, then the problem reduces to allocating G_{task_i}'s such that more message pass between neighbouring processors. Both problems in general are known to be NP-complete [7].

In this paper, we design and analyze a set of algorithms that attempt to achieve the goals of load-balancing and communication minimization for a special class of applications, namely those that can be represented by tree or linear task graphs, on a shared

memory architecture where $w(l_i)$ is same for all i's (i.e., the interconnection network is based on crossbar, shared bus, or multistage network, a unique characteristics of shared memory architecture. Our choice of this special class of problems is motivated by their wide practical applications and theoretical significance as described in the next paragraph. Whereas, the reasons for considering shared memory architecture are two-fold. First, as one of the two major architectural paradigms of multiprocessing it has great practical as well as theoretical importance, as evidenced by the intensive research activities about and commercial products of it. Second, due to it's symmetry and uniformity in network latency, the mapping $\mathbf{M}$ of $\mathbf{P}$ onto G_{arch} becomes trivial and straightforward.

A great number of computational problems in image processing, signal processing, generic algorithms, and scientific and engineering computing are naturally structured for pipelined, or iterative (parallel) computation or divide-and-conquer in nature [4, 9]. For instance, a problem, such as image/signal processing or generic algorithm, that requires a number of stages or iterations of sub-solution for its final solution can be viewed as being processed by a pipeline of sub-solutions. Thus, a sequence of such problems (possibly with different input parameters) can be "fed" to the pipeline and keep all stages busy. On the other hand, numerical methods for some scientific/engineering problems, such as partial differential equation, decompose the problem into strips of grid points of simple iterative calculations where each strip needs data from neighbouring strips for computation. In both cases, the parallel computation and communication requirement give rise to a chain-like (or linear) graph, where a node represents a collection of computation, called a task, and an edge signifies data-dependence or communication between two tasks. Similarly, algorithms and computations of divide-and-conquer nature form tree type structures, thus most appropriately represented by tree task graphs. It is common that the task graph representing a problem has more tasks than there are processors in a multiprocessor system. Furthermore, the computation and/or communication requirements may vary from task to task and among pairs of tasks.

Some papers closely related to our work have been published in recent years. Here we discuss about most relevant past works only. In his 1988 paper Bokhari [5] solved the problem of partitioning a linear task graph for linear array architecture and host-satellite architecture. He considered the problem for both homogeneous and non-homogeneous processors and gave partitioning algorithms for bottleneck minimization. It is to be noted that Bokhari made no attempt to minimize the bandwidth requirement which is very important for efficient execution of a parallel program on shared memory architecture. The time complexity of Bokhari's algorithm is fairly high. The algorithm runs in $O(n^3m)$ time, where n is the number of nodes in the linear task graph and m is the number of processors in the linear array multiprocessor. Bokhari's bottleneck minimization problem takes polynomial time when the task graph is a tree and target architecture is single host multiple (identical) satellite system. Whereas bandwidth requirement minimization problem is NP-complete for tree task graph and indirect interconnection network architecture (see Theorem 1 of Section 2.3).

Nicol and O'Hallaron made some improvement over Bokhari's algorithm in their 1991 paper [11]. They suggested an improved algorithm that solves the linear task graph partitioning problem in $O(n^2m)$ time for linear array architecture. Further, they solved the problem for homogeneous processors and homogeneous communication links in $O(mn \log n)$ time under constraints of bounded weights (module execution weights are bounded below and communication link weights bounded above). Further, they solved the problem of partitioning a linear task graph on shared memory architecture. The time complexity of their algorithm is $O(n \log n)$ and the space complexity is $O(n)$. Hansen and and Lih solved the partitioning problem for linear task graph and linear array architecture in $O(m^2n)$ time [8]. Though the time complexity achieved is no better than Nicol and O'Hallaron's algorithm, their approach is different, more lucid and easy to program.

We design polynomial time sequential algorithms for solving the problems of bottleneck minimization and processor minimization problems respectively, for tree task graphs executing on shared memory multiprocessors. To the best of our knowledge, they are proposed for the first time for its type. Then we present an improved algorithm for bandwidth minimization problem, applicable to linear task graphs executing on shared memory architecture. This algorithm has a time complexity of $O(n + p \log q)$ $(q \leq p \leq n$, where n is the number of tasks in the task graph) and space complexity of $O(n)$. This is in contrast to $O(n \log n)$, the time complexity of the best know algorithm of the type in the literature. We have also identified cases where our algorithm will run in linear time on the average. The paper is organized as follows. In section 2, the problems under consideration are formulated and a set of three polynomial algorithms are developed to solve the problems and their complexities analyzed. In Section 3 we present applications of our algorithms. Finally, concluding remarks are given in Section 4.

2 Partitioning Algorithms

We formulate the problem as the following graph theoretic problem. Given the task graph $G = (V, E)$, partition it satisfying the following two conditions.

1. **Execution time bound.** The sum of the vertex weights of all connected components after partition should be less than or equal to some K.
2. **Bottleneck minimization.** The maximum of the sum of the weights of the crossing edges between any two components should be minimum.

However, a partition satisfying these two conditions may not give a load balanced partition. Therefore, to obtain a balanced partition we need to add one more condition to the existing condition set. A good partition should satisfy one of the following conditions in addition to the time boundedness and bottleneck conditions.

1. **Processor minimization.** The number of connected components should be minimum.

2. **Bandwidth minimization** The sum of the weights of crossing edges should be minimum.

2.1 Bottleneck Minimization

For a general graph the bottleneck minimization problem is a variant of the graph partitioning problem. Graph partitioning problem is known to be NP-complete. However, in this paper we shall give a polynomial time algorithm for bottleneck minimization for tree task graph under the constraint that a connected component is assigned to each processor.

It can be clearly seen that for a tree task graph the bottleneck minimization problem is equivalent to the following problem. Given a tree $T = (V, E)$ find $S \subseteq E$ such that

1. The vertex weight of no connected component of $T - S$ exceeds K, and

2. $\max_{e \in S} \delta(e)$ is minimum for all $S \subseteq E$ satisfying condition 1, where δ is the edge weight function.

Algorithm 2.1 computes a solution to the bottleneck minimization problem in $O(n^2)$ time. The algorithm puts all least weight edges in S first. Then it checks whether the vertex weights of all connected components of $T - S$ are less than or equal to K. If the condition is satisfied, the algorithm stops giving output S. If the condition is not satisfied the algorithm continues by adding least weight edges from the remaining edges to S.

Algorithm 2.1

1. Sort all the edges of T in the order of increasing weights.
   Let the sorted list be $e_1, \ldots, e_{n-1}$.
2. for $i \leftarrow 1$ to $n-1$ do
   $S \leftarrow S \cup \{e_i\}$;
   if vertex weights of all connected components of $T - S \leq K$
   Output S and exit;

Proof of correctness. Let S' be a solution to the problem and e_s be the maximum weight edge in S'. Note that S' is a subset of $\{e_1, \ldots, e_s\}$ and the vertex weights of all connected components of $T - S'$ are less than or equal to K. Therefore, algorithm 2.1 will give a subset of $\{e_1, \ldots, e_s\}$ as output. Thus the proof.

2.2 Processor Minimization

The bottleneck minimization algorithm may fragment the task graph into unnecessarily many small components. High fragmentation may cause unbalanced load leading to under-utilization of processors. To prevent processor under-utilization, it is necessary to minimize number of processors required. The bottleneck minimization algorithm divides the tree task graph into several connected components. Note that there may be at most one edge between two connected components. Therefore, if all vertices belonging to one connected component are lumped together forming super-nodes (the weight of a super-nodes is sum of the weights of the vertices belonging to it), then the resulting graph is still a tree. Now the problem of processor minimization reduces to the problem of finding an edge cut S for the new task graph T such that, 1) the vertex weight of any connected component of $T - S$ is bounded by K and 2) the number of connected components of $T - S$ is minimum.

Given a tree (or for that matter, forest) removal of one edge leads to exactly one more connected component. Therefore, minimizing the number of components in $T - S$ is same as minimizing $|S|$. We have adapted an algorithm presented in [1] for this purpose.

If the task graph T is a star graph, then minimizing $|S|$ is easy. If the sum of the vertex weights of T is less than or equal to K then $S = \emptyset$. If the sum is greater than K then sort the leaves in increasing order of weights. Then continue to prune the leaves from the beginning of the list until the weight of the connected component containing the center of the tree is less than or equal to K. The algorithm presented here is a generalization of this idea. The algorithm is demonstrated by an example in figure 1.

Algorithm 2.2

proc_min(T : *weighted tree*, K)

1. If T has one vertex, return($\emptyset$);
2. Choose an internal node v of T such that v is adjacent to at most one internal node;
3. $W \leftarrow$ sum of all leaves adjacent to v + weight of v;
4. If $W \leq K$ then
   form a tree T' by pruning all leaves adjacent to v. Change the weight of v in T' to W. return(*proc_min*(T', K)).
5. Sort the leaves adjacent to v in decreasing order of weights. Let the sorted list be $v_1, \ldots, v_s$ with corresponding weights $w_1, \ldots, w_s$. Let e_i be the edge between v and v_i. Find minimum r such that $W - \sum_{i \leq r} w_i \leq K$. Form a tree T' by pruning all leaves adjacent to v. Change the weight of v in T' to $W - \sum_{i \leq r} w_i \leq K$.
   return($\{e_1, \ldots, e_r\} \cup$ *proc_min*(T', K)).

The recursive routine algorithm 2.2 will be executed once for each internal node of T. If the degree of an internal node v is $d(v)$ then the routine will take $O(d(v) \log d(v))$ time for the processing involved with v. Therefore, the time complexity of the algorithm is $O(\sum d(v) \log d(v))$. For a tree $\sum d(v) = O(n)$. Hence, $O(\sum d(v) \log d(v)) = O(\log n \sum d(v)) = O(n \log n)$. Therefore, the time complexity of the algorithm is $O(n \log n)$.

2.3 Bandwidth Minimization

The bandwidth minimization problem subject to load balancing constraint is NP-complete even when

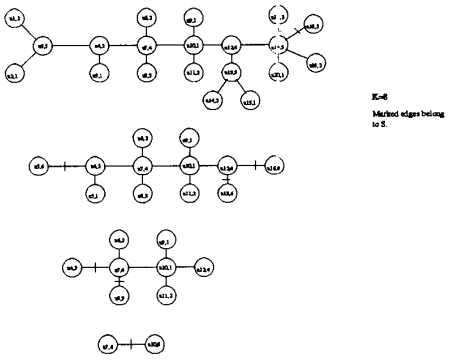

Figure 1:

the task graph is a very simple tree like star. Here we offer a simple proof of it. Interested readers may refer to [12] for some similar NP-completeness proofs.

Theorem .1 *Given a tree $T = (V, E)$ with vertex weight function $\omega : V \rightarrow R^{\geq 0}$ and edge weight function $\delta : E \rightarrow R^{\geq 0}$ and two positive numbers k_1 and k_2. The problem of finding an edge cut $S \subseteq E$ such that $\delta(S) \leq k_1$ and the vertex weights of all connected components of $T - S$ are less than or equal to k_2 is NP-complete.*

Proof. The NP-completeness proof is by reduction to 0-1 knapsack problem [7].

Consider the 0-1 knapsack problem with weights $w_1, \ldots, w_r$, profits $p_1, \ldots, p_r$, minimum profit k_1 and knapsack capacity k_2. Construct a star graph $T = (V, E)$, where $V = \{u, v_1, \ldots, v_r\}$ and $E = \{e_i \mid e_i = (u, v_i)\}$. Construct the weight functions as follows, $\omega(u) = 0$, $\omega(v_i) = w_i$ and $\delta(e_i) = p_i$ for $i = 1, \ldots, r$.

Let $S \subseteq E$ be an edge cut such that $\delta(S) \leq \sum p_i - k_1$ and the vertex weights of all connected components of $T - S$ are less than or equal to k_2. Let $I = \{i \mid e_i \notin S\}$. Note that,

$$\sum_{i \in I} w_i \leq k_1, \quad \sum_{i \in I} p_i \geq k_2.$$

Therefore I is a solution to the 0-1 knapsack problem.

Similarly, given a solution I' for the 0-1 knapsack problem, it may be shown that $S' = \{e_i \mid i \notin I'\}$ is a solution to the bandwidth minimization problem.

□

The above proof may be extended for the case when the vertex weights are strictly positive. Further, the problem may be proved to be NP-complete for most of the common task graphs using similar arguments. However, the bandwidth minimization problem is polynomial for linear task graphs. Here we present an algorithm to solve the problem for linear task graphs.

Let $P = (V, E)$ be a path, where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_i \mid e_i = (v_i, v_{i+1})\}$. Let $\alpha : V \rightarrow \Re^+$ and $\beta : E \rightarrow \Re^+$ be respectively vertex weight and edge weight functions. For convenience of notation, let $\alpha_i = \alpha(v_i)$ and $\beta_i = \beta(e_i)$. Given a non-negative number $K > \max_i \alpha_i$, find an edge cut $S \subseteq E$ of P, such that,

1. Sum of the vertex weights of any connected component of $P - S$ is less than or equal to K, and
2. S is of minimum weight among all subsets of E satisfying (1).

For any set $S \subseteq E(P)$ let $\beta(S)$ denote the weight of S, i.e, $\beta(S) = \sum_{i \mid e_i \in S} \beta_i$.

Let a subpath of vertex weight more than K be called a *critical subpath*. It is to be noted that every connected component for $P - S$ is a subpath of P. Therefore, if $S \subset E$ contains at least one edge from every critical subpath, then $P - S$ may not have any connected component of vertex weight more than K. Conversely, if $P - S$ does not have any connected component of vertex weight bigger than K, then S must contain at least one edge from every critical subpath.

To solve the problem, the following approach is taken.

1. Consider all critical subpaths of P.
2. Find S, a minimum weight subset of E, such that for any critical subpath P_1, S has an edge from P_1. In other words, S has non-null intersection with the edge sets of all critical subpaths.

As P is a path of length n, there are $\binom{n}{2}$ possible subpaths of P. Therefore, potentially there are $O(n^2)$ critical subpaths. However, for the problem under consideration all possible subpaths of vertex weight bigger than K need not be used in computation. If a critical subpath is subpath of another critical subpath, then former one is called *prime subpath* and later subpath is called *dominated subpath*. It is easy to see that the solution of part two will not change if only the prime subpaths are considered. The possible number of prime subpaths is bounded above by $n - 1$. Let p denote the exact number of prime subpaths. It is easy to see that all p prime subpaths may be computed in linear time.

The second part of the problem is a special case of the weighted hitting set problem. The weighted hitting set problem is a generalization of the unweighted hitting set problem which may be defined as follows.

Definition 2.1 *Given $A_1, \ldots, A_r \subseteq U$, find $B \subseteq U$ such that*

1. *$A_i \cap B \neq \emptyset$ for $i = 1, \ldots, r$, and*
2. *$|B| \leq |B'|$ for all $B' \subseteq U$, satisfying (1).*

It is known that hitting set problem is NP-hard even when $|A_i| \leq 2$ $\forall i$ (see [7]). Therefore, the weighted hitting set problem is also NP-hard. However, the weighted hitting set problem under consideration is more structured than a general weighted hitting set problem. More specifically, in our problem

the sets are the edge sets of subpaths of a path P. Because the edges are ordered in P, therefore all edges belonging to any subpath occurs consecutively. This property helps to contain combinatorial explosion and leads to a polynomial time algorithm.

Let $P_1, \ldots, P_p$ be p prime subpaths. Let the edge set of P_i be $E(P_i) = \{e_{a_i}, \ldots, e_{b_i}\}$. In other words, when the edges of P are ordered as $e_1, \ldots, e_{n-1}$; then P_i contains all edges between a_i and b_i, both inclusive. Further, it may be assumed w.o.l.g, $a_1 < \cdots < a_p$, i.e, P_i's are ordered according to the increasing order of left end. It is to be noted that when the subpaths are ordered in the increasing order of left end, an edge belongs to several consecutive subpaths. Alternatively, every edge e_j belongs to subpaths $P_{c_j}, \ldots, P_{d_j}$. Therefore, beginning from P_1, P_{c_j-1} is the last subpath such that none of $P_1, \ldots, P_{c_j-1}$ contains e_j. Let us denote $c_j - 1$ by γ_j. Formally,

$$\gamma_j = \max_{e_j \notin P_1, \ldots, P_i} i.$$

Let S_i be the minimum weight edge cut of P such that $S_i \cap E(P_j) \neq \emptyset$ for $j = 1, \ldots, i$; $i = 1, \ldots, p$. That is, S_i is the solution of the problem when only first i subpaths are considered. To solve the problem totally we need to compute S_p. It is easy to see that $S_1 = \{e_s\}$ where e_s is the minimum weight edge of P_1. Then the relation between S_i's may be expressed as a recurrence relation. $S_1 = \{e_s\}$ where, $\beta_s = \min_{a_1 \leq j \leq b_1} \beta_j$ and $S_{i+1} = \{e_s\} \cup S_{\gamma_s}$ where $\beta_s + \beta(S_{\gamma_s}) = \min_{a_{i+1} \leq j \leq b_{i+1}} \beta_j + \beta(S_{\gamma_j})$.

Computing the recurrence relation in this naive way will take $O(\sum_{i=1}^{p} |P_i|)$ time, which may be as high as $O(np)$. It is to be noted that if two edges e_i and e_j belong to exactly same subpaths under consideration, then the edge with higher weight will never belong to any S_r's. A list of non-redundant edges may be prepared in $O(n)$ time. It may be noted that there may be at most $2p - 1$ non-redundant edges.

Nicol's algorithm solves the problem in $O(n \log n)$ time [11]. Obviously, the naive implementation of our algorithm does worse than that. We presented the naive version for ease of understanding. Now we present a different implementation that runs in $O(n + p \log q)$ time and $O(n)$ space where $n \geq p \geq q$. Further, we have identified cases where our algorithm will run in linear time on the average.

2.3.1 An $O(n + p \log q)$ time Algorithm

To begin with the new implementation, we compute the prime subpaths in terms of non-redundant edges only. Without loss of generality assume that $e_1, \ldots, e_r$ are non-redundant edges. where $r \leq \min(n, 2p - 1)$. $P_1, \ldots, P_p$ are prime subpaths where $E(P_i) = \{e_{a_i}, \ldots, e_{b_i}\}$. Let, $W_i = \beta_i + \beta(S_{\gamma_i})$. At times we shall refer to W_i's as W-values. Note that $W_i = \beta_i$ for $a_1 \leq i \leq b_1$.

The goal of the algorithm is to compute minimum of W-values for edges belonging to each prime subpath. At the beginning of the algorithm all W-values are not known. Only after processing the edges belonging to P_1 and computing the minimum of W-values for those edges, we can know W_{b_1+1}. Therefore, in our algorithm we process edges from left to right and keep on updating the values of S_i's. Let e_i belong to q_i prime subpaths, $1 \leq i \leq r$, where $r \leq \min(2p - 1, n - 1)$. Therefore, if the updating is done in the naive way, for each e_i we shall have to update q_i S's. It is to be noted that $\sum q_i$ may be as high as $O(p^2)$, which, in many cases, is worse than currently known $O(n \log n)$ algorithm. To bring down the cost of updating to $O(\log q_i)$ we use a special data structure.

The data structure used here is a $p \times 4$ array called TEMP_S. It is an implementation of a queue from which elements may be removed from both the head and tail. TOP and BOTTOM are two pointers pointing to the head and tail of the queue. After processing each edge, TEMP_S contains the information about all the prime subpaths the edge processed belongs to.

After processing i-th edge, TEMP_S will contain the following information. The first two columns (L and R column) of each row will point to a range over the indices $1, \ldots, p$. Each prime subpath with index in the range has same minimum W-value after processing e_i. The minimum W-value is kept in the third column (W column). The fourth column (S column) contains the edge, e_j, for which the minimum W-value is achieved and S_{γ_j}.

Consider the value of TEMP_S after initialization, i.e, after processing all edges belonging to P_1. In this case P_1 contains $e_1, \ldots, e_4$. The first row of TEMP_S shows that after processing e_4, the minimum W-value for P_1 to P_2 is 4 and it is achieved for e_2. Further, at this stage, the minimum W-value for P_31 to P_4 is 5 and it is achieved for e_4.

The process of initialization is as follows. Initialize TEMP_S$(3, b_1)$ by W_{b_1}, TEMP_S$(4, b_1)$ by $\{e_{b_1}\}$, and TEMP_S$(2, b_1)$ by b_1. Let x be the maximum index less than b_1 such that $W_x < W_{b_1}$. Initialize TEMP_S$(1, b_1)$ by $x + 1$, TEMP_S$(3, b_1 - 1)$ by W_x, TEMP_S$(2, b_1 - 1)$ by x and TEMP_S$(4, b_1 - 1)$ by $\{e_x\}$. Continue this way until all edges in P_1 is processed. See step 1 of the algorithm in appendix for a formal description.

Few things are to be noted here. First, the length of the queue TEMP_S never exceeds q_i after processing e_i. Second, the third column (W column) in TEMP_S will always remain sorted in increasing order. Therefore, any search on W column can be done in $O(\log q_i)$ time.

All edges not in P_1 will be processed from left to right in similar way. Before processing e_i, check with TEMP_S whether it contains information about any prime subpath that does not contain e_i. Note that there may be at most one such subpath and that subpath must be pointed by the L-column of the TOP row. Therefore, the checking can be done in $O(1)$ time. If there is such a path, the processing for that path is complete and the W and S columns of the TOP row contain the minimum W-value and S-value for that path. Store the S and W value for the path. Discard the path from TEMP_S. It is done by increasing the L column of TOP row by one. Check that whether TOP row consists any valid range, i.e, if the value of L column exceeds the value of R column. If so, delete

the row itself.

At this stage all prime subpaths belonging inside the ranges pointed by L and R columns contain e_i. Compute W_i. To determine for which prime subpaths the minimum W-value needs to be changed, perform a binary search on the W column of TEMP_S. The outcome of the search will give a row index (say α), such that TEMP_S$(\alpha, 3) \geq W_i$ and TEMP_S$(\alpha - 1, 3) < W_i$. If no such α exists, no update is necessary. The minimum W-values for all prime subpaths pointed by α-th and subsequent rows need to be updated. Delete all these rows and add a new row pointing to all prime subpaths pointed by deleted rows. Note that this can be done in $O(1)$ time. The W column and S column of the new row will contain W_i and $\{e_i\} \cup S_{\gamma_i}$ respectively.

TEMP_S needs to be updated again, if any new prime subpath begins with e_i. If so, that information needs to be included in TEMP_S. If W value of e_i is already in the table (W column, BOTTOM row) then increase the value of R column BOTTOM row by one. Otherwise, create a new row pointing to the new subpath and containing the W and S values.

The formal description appears in appendix.

2.3.2 Merits of the New Algorithm

Note that the algorithm iterates $O(p)$ times and each step takes $(\log q_i)$ time. Therefore, the time complexity of the algorithm is $O(p \log q)$ time, where $q = \sum q_i / r$ and r is the number of non-redundant edges.

The best known algorithm known so far ([11]) runs in $O(n \log n)$ time. Whereas our algorithm runs in $O(p \log q)$ time. It is easy to see that $n \geq p \geq q$. Therefore, the worst case complexity of our algorithm is at least as good as the best known algorithm. However, it is possible to construct pathological cases where $n \log n = O(p \log q)$. Hence, we do not claim much improvement in the worst case.

We have done extensive simulation to obtain the relation between n, p, q, K, $p \log q$ and maximum vertex weight (maximum module execution time). The outcome of those simulations has been summarized in figures 2. From those figures it can be seen that for given n, $p \log q$ may be very low in many cases (particularly for high and low K). Therefore, it may be said that our algorithm exploits the nature of data and runs in considerably less time if data permit, while retains the worst case performance at least as good as the best known current algorithm. From the figures 2 it may be noticed that the maximum value of $p \log q$ is much less than $n \log n$. Therefore, we expect a constant time improvement even in the worst case.

Note that q_i is bounded by the length of the first prime subpath in which e_i belongs to. Therefore, the expected q is bounded by the expected lengths of prime subpaths in terms of non-redundant edges. If the vertex weights are distributed uniformly over the range $[w_1, w_2]$, the average length of prime subpaths will be bounded by $2K/(w_1 + w_2) \leq K/w_2$. That is, if K/w_2 is bounded by some constant, then q also will be bounded by the same constant on the average.

Another factor about our algorithm needs to be noticed. The $\log q$ factor in the time complexity occurs for the search on the TEMP_S array. At the i-th step, TEMP_S contains information about q_i subpaths. Therefore, number of entries in TEMP_S can never exceed q_i at i-th step. However, TEMP_S contains exactly one entry for all subpaths with same minimum W value. Therefore, in most of the cases, TEMP_S will contain less than q_i entries at i-th step, except when W values will occur sorted in ascending order. If we assume that W values are occurring at random (with respect to relative order), the argument in appendix B shows that the average length of TEMP_S at i-th step will be $O(\log q_i)$. Therefore, our algorithm will run in $O(p \log \log q)$ time on the average. However, more often than not, W values will have a tendency to grow towards end. Hence, the expected length of TEMP_S array will be more than $O(\log q_i)$. But how bigger it will be is not known. Further, the fact that W values have a tendency to grow towards end may be utilized to devise a k-ary search (k depending on the distribution of vertex weights and K) instead of binary search. If properly designed such a search may reduce the search time by a log factor. We are still working on these problems.

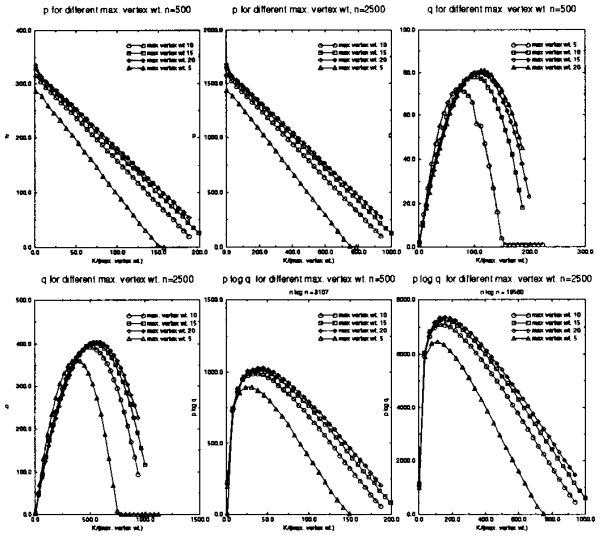

Figure 2:

3 Applications

The algorithms developed in this paper deals with a special class of applications whose task graphs are of linear or tree type, as shown in Figure 1, and assume that the underlying architecture is shared memory architecture. The unique characteristics of shared memory architecture that it's network latency is symmetric and uniform renders a straightforward mapping of the optimally partitioned graph onto the available processors in the multiprocessor, provided that the number

of processors is greater than or equal to that of the partitions. In this section we discuss two examples to which our algorithms may be applicable.

Real-Time Computing The general nature of real-time computing is that a task must be completed by a given deadline. In real world applications, failures to meet the deadline can result in damaging, if not catastrophic, consequences. One way to meet the time constraint is to explore the hidden parallelism in the application, that is, to partition the problem into concurrently executable subproblems. In doing so, however, one must also counter-balance the adverse impact of such factors as interprocess communication cost and reliability that are inherent in parallel and distributed processing. Consider a real-time task **T** to be executed on a shared memory multiprocessor, with the following constraints.

- The time deadline for computing **T** is k, a positive constant.
- **T** can be maximally divided into a set of subtasks $T = \{t_1, t_2, ..., t_n\}$ and there is a data dependency dp_i between t_i and t_{i+1} for $1 \leq i \leq n-1$.
- $w(t_i) \leq k$, $1 \leq i \leq n$, is the total processing time required for task t_i including computation and communication.
- $w(dp_i)$, $1 \leq i \leq n-1$, is a weight associated with data dependency dp_i reflecting the communication/resource cost (i.e., traffic/resource demand on the interconnection network) and/or reliability factor (i.e., some data are more sensitive and less tolerant to network noise/fault than others).

These constraints mandate that the problem be partitioned in a way such that: 1) all subproblems must be completed within time k, 2) impact of network cost and noise must be minimized and 3) the highest traffic demand of a single processor on the network must be minimized. More specifically, T is to be partitioned into disjoint sets $T_1 = \{t_1, ...t_{i_1}\}$, $T_2 = \{t_{i_1+1}, ..., t_{i_2}\}$, ..., and $T_p = \{t_{i_{p-1}+1}, ..., t_n\}$ such that, for all $1 \leq j \leq p$, $w(T_j) = \sum_{l=i_{j-1}+1}^{i_j} w(t_l) \leq k$ and $\sum_{m=1}^{p-1} w(dp_{i_m})$ is minimum and $\max_{m=1}^{p-1} w(dp_{i_m})$ is minimized.

It is not difficult to see that such an optimal partition can be readily constructed by applying the algorithms of Section 2. The resulting partition, with p connected components, can be directly mapped to the shared memory multiprocessor. The partitioning and mapping process is illustrated in Fig. 3.

Distributed Discrete Event Simulation One important parallel and distributed processing application is discrete event simulation [10]. In a distributed discrete event simulation, the state of each process changes upon the occurrence of an event, which often exists in the form of a message passed from another process, at a discrete point of time. A process represents an integral part of the simulated system, for instance, a logic gate, a basic circuit, or a submodule of a logic system. Thus, such a simulation can be represented by a task graph, as defined in Section 1, where an edge links two processes which need to pass messages to each other directly. In the case of logic circuit simulation, an edge also represents a physical link between the two processes (or gates). A weight is associated with each process to indicate its processing requirement, whereas the number of messages needed to be passed between two processes is signified by a weight associated with the connecting edge. Both quantities in general are determined by the requirement of the simulation, the task structure, and the techniques used to simulate. When some simulation techniques are used [10], they are often fixed regardless where the processes are allocated.

While there has been a significant amount of research on distributed simulation in general and distributed logic simulation in particular, one problem was often omitted in previous studies. That is, that of how to strategically partition a simulation task graph (e.g., a circuit) on a multiprocessor so that the load on all processors are balanced and the number of messages passed among processors is minimized. Due to the NP-Completeness of the general problem, most current partitioning strategies are based on heuristic solutions [6, 3, 2]. However, when the simulated system is circular or linear in nature or can be approximated by a linear task graph, such as a circular type logic circuit or network and assuming that the underlying simulating machine is shared memory architecture (a likely event in light of the large presence of shared memory multiprocessors in the form commercial products or research prototypes), the problem reduces to the special case for which our algorithms can be applied, giving rise to an efficient solution. More specifically, if the topological structure of the simulated system renders a linear process graph then the application of our algorithm becomes straightforward. Otherwise, for a more general system, we may first approximate the original system by generating a super-graph, which is linear, from the process graph, then apply the algorithm to the super-graph.

4 Conclusion

Strategically partitioning process graphs onto multiprocessors so as to balance the load while minimizing the communication overhead is of great importance. This is because it enables one to improve the quality of and to exploit potentials in parallel and distributed processing. Since the general problem is NP-complete, current solutions to the problem have been mostly based on heuristic algorithms. In this paper, we investigated a special case of the problem, namely, the one with tree and linear task graphs. We design and analyze an algorithm that solves the partitioning problems for this special class of applications. To our knowledge, the bottleneck minimization and processor minimization algorithms presented in this paper are first of their type. The bandwidth minimization algorithm proposed has an improved time complexity over the best known algorithm in the literature.

The algorithms developed, while limited in their scope of applications, have a number of advantages: The algorithms are simple and efficient; the resulting partition can be mapped straightforwardly to a common multiprocessor architecture, namely, shared

memory architecture; and more general cases may be approximated by generating a linear or tree supergraph of the original process graph.

Appendix A

Algorithm 4.1

1. Initialization.

BOTTOM←TOP← b_1; TEMP_S$(3, b_1) \leftarrow W_{b_1}$;
TEMP_S$(2, b_1) \leftarrow b_1$; TEMP_S$(4, b_1) \leftarrow \{e_{b_1}\}$;
for $i \leftarrow b_1 - 1$ downto a_1 do
  If W_i <TEMP_S$(3, TOP)$ then
    TEMP_S$(1, TOP) \leftarrow i + 1$; $TOP \leftarrow TOP - 1$;
    TEMP_S$(4, TOP) \leftarrow \{e_i\}$;
    TEMP_S$(3, TOP) \leftarrow W_i$; TEMP_S$(2, TOP) \leftarrow i$;

TEMP_S$(1, TOP) \leftarrow 1$;
TEMP_S$(3, i) \leftarrow -\infty$ for all $i < TOP$;
TEMP_S$(3, i) \leftarrow \infty$ for all $i > BOTTOM$;

2. for $i \leftarrow b_1 + 1$ to r do
  $j \leftarrow$ TEMP_S$(1, TOP)$;
  if $b_j < i$ then
    $S_j \leftarrow$ TEMP_S$(4, TOP)$;
    TEMP_S$(1, TOP) \leftarrow$ TEMP_S$(1, TOP) + 1$;
    if TEMP_S$(1, TOP)$ >TEMP_S$(2, TOP)$
      $TOP \leftarrow TOP + 1$;

2a. Find s such that TEMP_S$(3, s) \geq W_i$
  and TEMP_S$(3, s - 1) < W_i$;

  TEMP_S$(3, s) \leftarrow W_i$; TEMP_S$(4, s) \leftarrow \{e_i\} \cup S_{\gamma_i}$;
  TEMP_S$(2, s) \leftarrow$TEMP_S$(2, BOTTOM)$;
  $j \leftarrow$TEMP_S$(2, s)$;
  if $a_{j+1} = i$ then
    TEMP_S$(2, s) \leftarrow j + 1$;
  if $s > BOTTOM$ then
    TEMP_S$(1, s) \leftarrow$TEMP_S$(2, s)$;
  $BOTTOM \leftarrow s$;

Solution S_p is TEMP_S$(4, BOTTOM)$;

Appendix B

Let e_i belongs to prime subpaths $P_x, \ldots, P_y$. Let the edges with indices less than or equal to i belonging to $P_x, \ldots, P_y$ be $e_j, \ldots, e_i$. Let the minimum W-value for $e_j, \ldots, e_i$ be W_{j1}. If W-values are distributed at random, $j1$ will at the middle of the range $j, \ldots, i$ on the average. There will be exactly one entry in TEMP_S for all prime subpaths containing e_{j1}. By the similar argument there will be exactly one entry corresponding to minimum W-value over the range $e_{j1+1}, \ldots, e_i$. By repeating this argument we show that there will be $O(\log i - j + 1)$ entries in TEMP_S on the average. Note that $i - j + 1 \leq 2(y - x + 1) = 2q_i$. Therefore, there will be $O(\log q_i)$ entries in TEMP_S on the average.

References

[1] K. S. Bagga et al., "Some Bounds and an Algorithm for the Edge-Integrity of Trees", To appear *J. Comb. Math. Comb. Comp.*

[2] E. Barnes, "An Algorithm for Partitioning the Nodes of a Graph", *SIAM J. DISC. ALG. METH*, Vol. 3, No. 4, Dec 1982, pp. 541-550.

[3] E. Barnes and A. Vannelli, *A New procedure for Partitioning the Nodes of a Graph*, Working Report No. 85-2, University of Toronto, 1985.

[4] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation-Numerical Methods*, Prentice-Hall, 1989.

[5] S. H. Bokhari, "Partitioning Problems in Parallel, Pipelined and Distributed Computing", *IEEE Trans. Comp.*, Vol. 37, No. 1, Jan. 1988, pp. 48-57.

[6] C. Fiduccia and R. M. Matteyses, "A Linear-Time Heuristic for Improving Network Partitions", *Proc. Design Automation Conf., IEEE*, 1982, pp. 175-181.

[7] M. Garey and D. Johnson, *Computers and Intractability - A Guide to the Theory of NP-completeness*, W. H. Freeman and Co. New York.

[8] P. Hansen and K-W Lih, "Improved Algorithms for Partitioning Problems in Parallel, Pipelined and Distributed Computing", *IEEE Trans. Comp.*, Vol. 41, No. 6, Jun. 1992, pp. 769-771.

[9] J. Ja'Ja', *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.

[10] J. Mishra, "Distributed Discrete-Event Simulation", *ACM Computing Surveys*, Vol. 18, March 1986, pp. 39-65.

[11] D. M. Nicol and D. R. O'Hallaron, "Improved Algorithms for Mapping Pipelined and Parallel Computations", *IEEE Trans. Comp.*, Vol. 40, No. 3, Mar. 1991, pp. 295-306.

[12] S. Ray and J. S. Deogun, "Computational Complexity of Weighted Integrity", To appear *J. Comb. Math. Comb. Comp.*