Parallel Algorithms for the Positive Linear Programming Problem

by

Virat Agarwal Umang Sharan

B.Tech. Project Thesis

Under the Guidance of
Dr. Naveen Garg

With the
Department of Computer Science & Engg.
Indian Institute of Technology, Delhi

Acknowledgment

We would like to thank Dr. Naveen Garg for the time and effort that he has put in to help us with our project. Apart from providing us with the necessary motivation, he gave us his complete support and backed us in all our efforts. We are greatly indebted to him for whatever we have achieved in our project. Without his help and guidance this work would have not been possible.

Umang Sharan

Virat Agarwal

Certificate

This is to certify that the project titled "Parallel Algorithms for the Positive Linear Programming Problem" being submitted by **Umang Sharan**, entry number 2002142 and **Virat Agarwal**, entry number 2002144 to the Department of Computer Science & Engineering, Indian Institute of Technology, Delhi in the year 2006, as a partial fulfillment of the requirement for the degree of Bachelors of Technology, in Computer Science & Engineering is a record of bonafide work carried out by them under my supervision. The results obtained in this report have not been submitted to any other institution for the award of any other degree or diploma.

Dr. Naveen Garg
Department of Computer Science and Engg.
Indian Institute of Technology, Delhi

Abstract

We introduce a fast parallel approximation algorithm for solving the Positive Linear Programming Optimization Problem, i.e. the special case of linear programming problem where the input constraint matrix and constraint vector consist of positive entries. Papadimitriou and Yannakakis initiated the study of such problems in a framework of solving positive linear programs by distributed agents. We take their model further and turn attention to the trade off between the running time and the quality of the solution. Our algorithm draws from techniques developed by Bartal, Byers & Raz [1], Garg & Konemann [2], Luby, Nissan [4] with the intention to achieve maximum parallelization to get a faster algorithm.

Contents

List of Figures

1 Introduction

The positive linear programming optimization problem is the special case of the linear programming optimization problem where the input constraint matrix and constraint vector consist entirely of non-negative entries. We introduce an algorithm that takes as input the description of a problem and an error parameter ϵ and produces both a primal feasible solution and a dual feasible solution, where the values of these two solutions are within a $(1 + \epsilon)$ multiplicative factor of each other. Because the optimal values for the primal and dual problems are equal, this implies that the primal and dual feasible solutions produced by the algorithm have a value within ϵ (with respect to relative error) of an optimal feasible solution.

Achieving a global goal based on local information only is one of the key challenges when developing fast parallel and distributed algorithms. In k rounds of communication, a network node can only gather information about nodes which are at most k hops away. Not surprisingly, many global criteria such as obtaining a spanning tree cannot be met by a local algorithm i.e. by an algorithm whose time complexity is much smaller than the diameter of the network graph or even constant. But what can be computed locally? In a seminal paper, over a decade ago, Naor and Stockmeyer posed this question which is fundamental for the theory of distributed computing.

In a parallel environment, the main issue is not the round of communications but the running time required. Let m be the number of constraints and n be the number of variables. Then our algorithm can be implemented on a parallel machine using O(m + n) processors with $O(\log(m + n)/\epsilon)$ phases.

Packing and Covering problems [**?**] that can be formulated as linear programs using only non-negative coefficients and non-negative variables. Covering and packing problems [**?**] occur in a number of distributed applications. For Covering, the most prominent problem is of finding dominating sets (DS). A DS is a subset of the nodes of a graph such that for all nodes v, either v or a direct neighbor of v are in the DS. In computer networks, it is often desirable to have a DS in order to enable a hierarchical structure where members of the DS provide a service to their neighbors. A particular application can be found in the context of mobile ad-hoc networks. Mobile ad-hoc networks consist of wireless devices communicating without a stationary infrastructure. In order to improve the efficiency of flooding in particular and routing in general, it has proven beneficial to maintain a clustering. Routing is then carried out on the level of clusters. Usually, a clustering is obtained by computing a DS or one of the many variants such as a k-dominating set.

Positive linear programs are strong enough to represent several combinatorial prob-

lems. The first example is matching in a bipartite graph. From matching theory we know that relaxing the 0,1 program that defines the largest matching to a linear program does not change the optimal value. This program is positive, thus PLPP algorithm can be used to approximate the size of the largest matching in a bipartite graph.

The second example is that of a set-cover. In this case it is known that relaxing the 0-1 program that defines the minimum set cover to a linear program can decrease the optimum value by at most $\log(\delta)$ factor, where δ is the maximum degree of the set system. This program is again positive and thus can solved by the PLPP algorithm to produce a solution with approximation $(1+\epsilon)\log(\delta)$. A $(1+\alpha)$ approximation algorithm for fractional set cover problem is discussed in [**?**].

Previously, Ptolkin, Shmoys, Tardos have developed fast sequential algorithms for both the primal and dual versions of the PLPP. Though they introduce algorithms that are simple and efficient but they don't have fast parallel implementations.

2 Model

We consider the following model given by Papadimitriou and Yannakakis in which distributed agents generate approximate solutions to positive linear programs in the following standard form:

PRIMAL	**DUAL**
$\max X = \sum_{j=1}^{n} x_j$	$\min\ Y = \sum_{i=1}^{m} y_i$
$\forall i,\ \sum_j a_{ij} x_j \leq 1$	$\forall j,\ \sum_i a_{ij} y_i \geq 1$
$\forall j,\ x_j \geq 0$	$\forall i,\ y_i \geq 0$
$\forall i, j,\ a_{ij} \geq 0$	$\forall i, j,\ a_{ij} \geq 0$

Figure 1: Model for PLPP

We associate a primal agent with each of the n primal variables x_j and a dual agent with each of the m dual variables y_i. Each agent is responsible for setting the value of their associated variable. For any i,j such that $a_{ij} > 0$, we say that dual agent i and primal agent j are neighbors. By this definition, dual agents are neighbors of primal agents and vice versa.

In general however, positive linear programs are represented as:

$$\max X = \sum_{j=1}^{n} B_j x_j$$
$$\forall i,\ \sum_j a'_{ij} x_j \leq C_i$$
$$\forall i, j,\ a'_{ij} \geq 0$$

Figure 2: General Positive Linear Program

Clearly, this positive linear program can be converted to standard form by the local operation $a_{ij} = a'_{ij}/B_j C_i$. Another assumption that we make on the LP is that it is given to the algorithm in normalized form in which the a_{ij} are either 0, or satisfy $1/\gamma \leq a_{ij} \leq 1$. One can convert a problem in standard form to the normalized form simply by dividing all the constraints by $a_{max} = \max a_{ij}$, thereby setting $\gamma = a_{max}/a_{min}$, (where $a_{min} = \min a_{ij}$ s.t. $a_{ij} > 0$).

We focus on a PRAM model for parallelization. Parallel Random Access Machine (PRAM) is a popular model for writing parallel algorithms. It consists of a number of processors that have a common, shared memory. A parallel program is not very different from a sequential (imperative) program, but there is a special "for i pardo Pi" structure that allows for parallel execution of subprograms. A (n,m) PRAM consists of n processors and m memory locations where each processor is a Random Access Machine (RAM). All processors share the same memory and communicate through it. Computation proceeds in synchronized steps. During each step a processor may read an element from the shared memory to its local memory or may write an element from its local memory to the shared memory. Alternatively, it can perform an operation to locally held data. No processor will proceed with instruction i+1 before all other processors complete the i-th step. PRAM is a very simple model of parallel computation that helps algorithm designers to focus in the essence of parallelism.

As mentioned earlier, our program consists of n primal and m dual agents. Each agent is responsible for setting the value of its associated variable. We say that x is primal feasible if x satisfies all the constraints. $Opt(x)$ is a primal feasible solution s.t.,

$$\text{sum(Opt(x))} = \max_{x \in S} \text{sum(x)}$$
$$\text{where } S \text{ is } \{x | x \text{ is primal feasible}\}$$

We say that y is dual feasible if y satisfies all the constraints. $Opt(y)$ is a dual feasible solution s.t.

$$\text{sum(Opt(y))} = \min_{y \in S'} \text{sum(y)}$$
$$\text{where } S' \text{ is } \{y | y \text{ is dual feasible}\}$$

3 Previous Ideas

3.1 Bartal, Byers & Raz

Bartal's algorithm for approximating the solution of PLPP runs in phases, maintaining that at the end of each phase both primal and dual are feasible. The model used by this algorithm is the same as what we have described above. For all non zero a_{ij}, primal agent j is connected to dual agent i. Primal and dual agents are associated to primal and dual variables respectively. Throughout the algorithm, the values of x_i (dual) are dependent on the values of the neighboring y_j (primal) by exponential weighting function : $x_i = e^{\lambda_i \phi}/\psi$, where ψ is depends on the current phase and ϕ is chosen according to the approximation ratio, and λ_i is the value of the primal constraint. The proce-

```
Algorithm LP() {
    call Initialize()
    repeat until (ψ > ψ_f)
        call Round-Update()
        repeat until (min_j α_j ≥ 1)
            ∀j, if (α_j < 1) then y_j = y_j(1 + ε/φ)
            call Round-Update()
        }
        ψ = ψ(1 + ε)
    }
    ∀j, output y_j
}
```

Figure 3: Algorithm LP()

dure Initialize() is used to initialize the values of ψ_f, ψ, y_j, ϕ. α_j is the value of the dual constraint. Before the end of each phase the value of ψ is scaled up by a factor of $1 + \epsilon$, due to which some dual constraints are violated (corresponding to some primal variables). During the execution of the phase these primal variables are incremented and corresponding dual values modified (using Round-Update), till the time all dual constraints are satisfied. The algorithm proceeds for a fixed number of phases at the end of which we have a $1 + \epsilon$ approximate solution to the PLPP.

Intuitively, at the beginning of each phase the primal variables (corresponding to the dual constraints) which get selected are the ones whose length is less (or the value of dual constraint is less). These primal variables have lengths lying between $\frac{1}{(1+\epsilon)}$ and 1. One may also think that it makes sense to increment the values of these primal

variables as increments in them will have minimal affect in the values of the primal constraints at the same time incrementing the objective function.

The algorithm runs in $O(\ln n/\epsilon^2)$ phases - these iterations are fixed and are dependent on the initial value of ψ and final value ψ_f. The number of iterations in each phase is $O(\ln^2 n/\epsilon)$. Thus, the algorithm runs in time $O(\ln^3 n/\epsilon^3)$.

3.2 Garg & Konemann Packing LP

The problem considered here is the general Packing Positive Linear Programming Problem.

Packing Problem

$$\max \ c^T x$$
$$Ax \leq b$$
$$x \geq 0$$

where A, b, c are (mXn), $(mX1)$, $(nX1)$ matrices having all entries positive and $A(i,j)$ is atmost b_i. The dual of this problem is,

Covering Problem

$$\min \ b^T y$$
$$A^T y \geq b$$
$$y \geq 0$$

The algorithm proceeds in iterations, where in each iteration length of each primal variable is calculated as

$$length_y(j) = \Sigma_i \frac{A(i,j)y(i)}{c(j)}$$

Intuitively, primal variable for which the length is minimum (call is q) should be chosen in each iteration. This is because for this variable primal column evaluates to minimum which means that incrementing this variable will cost minimal increase in the values of primal constraints. Also, cost coefficient of this variable is high so that increase in this variable will result in high increase in value of primal objective function.

The maximum amount of increment in this primal variable is dependent on the minimum capacity edge i.e., row for which $b(i)/A(i,j)$ is minimum. Call this row p. The chosen primal variable is then incremented by an amount $b(p)/A(p,q)$ and its dual neighbors are modified as,

$$y(i) = y(i)(1 + \epsilon \frac{b(p)/A(p,q)}{b(i)/A(i,q)})$$

No dual variable is incremented by a factor greater than $(1 + \epsilon)$. Also the p^{th} dual variable is incremented by a factor exactly $(1 + \epsilon)$. This is where the approximation ratio is bounded by $(1 + \epsilon)$.

The algorithm runs till the value of the dual objective function becomes greater than 1. The time complexity of the algorithm includes the number of iterations till atleast one variable becomes $(1 + \epsilon)/b(i)$. This value is multiplied by m to account for the worst case when a whole cycle of dual variables occurs (variables that get incremented by $(1 + \epsilon)$) before a variable is chosen again.

On the basis of this algorithm we had an intuition that if we can choose more number of primal variables and correspondingly more number of dual variables in each iteration, maintaining the incremental constraints (to preserve analysis), we can achieve a better running time with the same approximation ratio.

3.3 Luby & Nissan

Luby and Nissan give a fast parallel approximation algorithm for the linear programming problem in special form which has been the basis for further work in the same area. They give an algorithm which runs on a parallel machine using $O(N)$ processors with a running time polynomial in $log(N)/\epsilon$ where N is the number of non zero coefficients associated with an instance of the problem. The input, in special form, is given as: For all (i, j), the input a_{ij} is such that either $a_{ij} = 0$ or $1/\gamma \leq a_{ij} \leq 1$, where $\gamma = \frac{m^2}{\epsilon^2}$.

PRIMAL	**DUAL**
$\max Z = \sum_i z_i$	$\min\ Q = \sum_j y_j$
$\forall j,\ \sum_i a_{ij} z_i \geq 1$	$\forall i,\ \sum_j a_{ij} q_j \leq 1$
$\forall i,\ z_j \geq 0$	$\forall j,\ q_i \geq 0$

Figure 4: Luby Nissan LPP

Given a problem instance in the special form, the algorithm developed by Luby & Nissan has the following properties:

$$\tau = \min\{\text{sum(z):z is primal feasible}\} = \max\{\text{sum(q):q is dual feasible}\}$$

On input $\epsilon > 0$ and a_{ij}, the output is a primal feasible solution

$$\text{z}=(z_1, ..., z_n)$$

and a dual feasible solution

$$\text{q}=(q_1, ..., q_m)$$

such that

$$\text{sum(z)}\leq\text{sum(q)}(1+\epsilon)$$

Since sum(z)$\geq \tau \geq$sum(q), this implies that sum(z)$\leq \tau(1+\epsilon)$ and sum(q)$\geq \tau/(1+\epsilon)$.

Algorithm PLP() {
  $(x_1, ..., x_n) \leftarrow 0$
  $\forall j, \alpha_j = \sum_i a_{ij} x_i, \alpha = \min_j\{\alpha_j\}, y_j = e^{-\alpha}$
  $\forall i, D_i = \sum_j a_{ij} y_j, D = \max_i\{D_i\}, B = \{i : D_i = D\}$
    repeat forever {
      $\forall i \in B, \uparrow x_i$ in the direction which makes $D_i \downarrow$ at the same rate
  }
  $\forall i, z_i \leftarrow x_i/\alpha$ and $\forall j, q_j \leftarrow (y_j)_{opt}/D_{opt}$
  output z_i and q_j
}

Figure 5: Algorithm PLP()

The algorithm computes the (1+ϵ) approximation to the actual solution in

$$O(\tfrac{log(n)log(m/\epsilon)}{\epsilon^4})$$

iterations. Each iteration can be executed in parallel using O(N) processors in time O(log(N)) on a EREW PRAM where N is the number of entries (i,j).

3.4 Rajgopalan & Vazirani

Given a universe U, containing n elements, and a collection $S = \{S_i : S_i \subseteq U\}$, of subsets of the universe, each associated with a cost c_S, the set cover problem asks for the minimal cost subcollection that covers all the elements of the universe. Set cover problem is shown to be a NP-Complete problem by Karp's Seminal Paper. This paper presents a RNC^3, $O(\log n)$ approximation algorithm for the set cover problem.

In the greedy approach set S such that $\min_S\{\frac{c_S}{|U(S)|}\}$ is added repeatedly to the set cover, where $|U(S)|$ is the number of uncovered elements of the set S. $cost(e)$ is the cost of covering e by the greedy algorithm. Thus, if e was first covered by set S, then

$cost(e) = \frac{c_S}{|U(S)|}$ at the time set S was chosen. The set cover problem can be viewed as a primal covering problem,

Covering Problem

$$\min \ \Sigma c_S x_S$$
$$\Sigma_{S \ni e} x_S \geq 1$$
$$x_S \geq 0$$

Define $value(e) = \min_{S \ni e} \frac{c_S}{|U(S)|}$. In the greedy algorithm $cost(e) = value(e)$ at the time an element is covered, and it chooses to add the set for which $\Sigma_{e \in U(S)} value(e) = c_S$. The greedy algorithm approximates the optimum by a factor of H_k, where k is the size of the largest set.

In the parallel approach what is needed is to relax the set selection criteria without significantly affecting the approximation ratio. In this algorithm cost effective sets are identified by choosing those that satisfy the inequality,

$$\Sigma_{e \in U(S)} \ value(e) \geq c_S/2$$

PARALLEL SETCOV

PreProcess.

Iteration:
For each uncovered element e, compute $value(e)$.
For each set S: include S in L if
(.) $\Sigma_{e \in U(S)} \ value(e) \geq c_S/2$
Phase:
(a) Permute L at random.
(b) Each uncovered element e votes for the first set S (randomly) s.t. $e \in S$.
(c) if $\Sigma_{e \ votes \ S} value(e) \geq c_S/16$, S is added to the set cover.
(d) If any set fails to satisfy (.) then that set is removed from L.
Repeat until L is empty.
Iterate until all elements are covered.

Figure 6: PARALLEL SETCOV Algorithm

The PARALLEL SETCOV satisfies the parsimonious accounting property with $\mu = 16$. If we choose cost(e) = 16*value(e) if e votes for S and S is added to set cover in the same

phase, then $\Sigma_e\ cost(e) \geq (cost\ of\ cover)$, thus the approximation ratio remains $O(H_k)$.

The algorithm advances in phases maintaining that in every subsequent phase the minimum value advances by atleast a factor of 2. By evaluating the lower and upper bounds on the minimum value after preprocessing, number of phases is atmost $O(\log n)$.

Call a set-element pair (S, e), $e \in U(S)$ good if $\deg(e) \geq \deg(f)$ for atleast three quarters of elements in $U(S)$. By probabilistic analysis, if e votes for S then the probability that f votes for S is greater than $1/2$ (this is due to the randomization step of L). This implies that is (S, e) is good then S should receive a lot of votes if e votes for S. By further analysis, it is shown that if (S, e) is good, e votes for S then with probability (1/15) S is picked. To find the number of iterations, the expected decrease in uncovered element set pairs (ϕ) is found. This comes out to be $\frac{1}{15}(no\ of\ good\ (S, e)\ pairs)$. Since a quarter of total (S, e) pairs are good, the decrease in ϕ is $\frac{1}{60}\phi$. Thus, the number of iterations is $O(\log nm)$ where initially $\phi = nm$. The algorithm runs in time $O(\log n \log nm \log nmR)$, where R is the largest set cost in bits.

In the next section, we try to model the voting technique described in this paper for the general Packing PLPP. We try to present a voting scheme based on a threshold to choose primal columns during each iteration whose value is to be incremented. We also introduce a notion of good primal columns satisfying the length threshold.

4 Our Work

In this report, we introduce a new approach for solving general Packing Positive Linear Programming Problems. We worked on several strategies based on the intuition developed while reading papers related to Linear Programming Problems. We also worked on analysis to see if our intuition matched theoretical proofs. We present below some of our ideas towards solving general packing PLPPs. While the first section gives two techniques towards modeling the Garg & Konemann [2] algorithm in a distributed environment, in the latter section we present an independent algorithm towards solving a PLPP in a parallel environment.

4.1 Voting and Threshold Based Approach

We adopted this approach while reading Rajgopalan & Vazirani's Paper on Primal-Dual RNC Approximation Algorithms for the Set Cover and Covering Integer Programs [5]. The main idea is to parallelize the minimum operation and let the columns select themselves locally based on a minimum number of votes it receives from its neighbors in any round. Hence, if a primal node receives a stipulated number of votes then it is selected otherwise not. The voting criteria works as follows: Each primal node

computes its length and broadcasts it to its neighbors. The dual neighbors then vote for the top n/4 of their neighbors giving m^n, m^{n-1} and so on votes in ascending order of primal lengths and broadcasts these to the respective primal nodes. Each primal node then sums the total number of votes it received in a particular round and if it is greater than some threshold V then it updates its value and the values of OF and dual neighbors accordingly.

$y \leftarrow \delta/b_i$
$D(0) \leftarrow m.\delta$
$x \leftarrow 0$
Algorithm ThresholdLP()

while (D(t) < 1) {
  $\alpha(y_t) = (\text{length})_{y_t}$
  foreach i s.t. $\alpha_i <$ T
    $f_k = f_{k-1} + \frac{c_q\ b_p}{A_{pq}}$
    $y_j = y_j\ (1 + \epsilon\ \frac{1}{\#\ columns\ selected}\ \frac{b_p/A_{pq}}{b_j/A_{jq}}), \quad \forall j \in N(i)$
  }
  T = T*(1 + ϵ)
}

Figure 7: Threshold based column selection

Similarly in the threshold based approach, each primal node calculates its length and picks itself if the length is less than some threshold T. The dual variables are then updated using normalized increments based on the total number of primal nodes selected in a round.

We implemented the threshold based algorithm (source code available in APPENDIX) to check for the feasibility of results because we found the threshold values in both techniques to be highly problem specific and we couldn't get a problem independent algorithm for the PLPP.

4.2 Our Algorithm

Both of the above techniques involved constants that were problem specific and hence time bounds independent of the input were not possible. Hence we propose a new approach for solving the PLPP. We modify the packing LPP as follows,

Primal Problem

$$\begin{aligned} \max\ & c^T x \\ & Ax \leq 1 \\ & x \geq 0 \end{aligned}$$

This problem can be rewritten as,

Primal Problem

$$\begin{aligned} \max\ & \alpha \\ & c^T x \geq \alpha \\ & Ax \leq 1 \\ & x \geq 0 \end{aligned}$$

On further analysis, since $y_i \geq 0$

$$\begin{aligned} & Ax \leq 1 \\ \Leftrightarrow\ & a_i^T x \ \leq\ 1 \qquad \forall i,\ 0 \leq i \leq m \\ \Leftrightarrow\ & y_i a_i^T x \ \leq\ y_i \quad \forall i,\ 0 \leq i \leq m \\ \Rightarrow\ & \Sigma_i y_i a_i^T x \ \leq\ \Sigma_i y_i \\ \Leftrightarrow\ & \frac{\Sigma_i y_i a_i^T x}{\Sigma_i y_i} \ \leq\ 1 \ \leq\ \frac{c^T x}{\alpha} \\ \Leftrightarrow\ & \frac{\alpha}{\Sigma_i y_i} \ \leq\ \frac{c^T x}{\Sigma_i y_i a_i^T x} \\ \Leftrightarrow\ & \frac{\alpha}{\Sigma_i y_i} \ \leq\ \frac{c_1 x_1 + ... + c_n x_n}{y_1(a_1^T x) + ... + y_m(a_m^T x)} \\ \Leftrightarrow\ & \frac{\alpha}{\Sigma_i y_i} \ \leq\ \frac{c_1 x_1 + ... + c_n x_n}{(a_1 y)x_1 + ... + (a_m y)x_n} \ \leq\ \max_j \ \frac{c_j}{\Sigma_i a_{ij} y_i} \\ \Leftrightarrow\ & \min_j \ \frac{\Sigma_i a_{ij} y_i}{c_j} \ \leq\ \frac{\Sigma_i y_i}{\alpha} \end{aligned}$$

In the algorithm proposed by Garg & Konemann [2], recall that length of a primal variable was calculated using,

$$length_j = \frac{\Sigma_i a_{ij} y_i}{c_j}$$

In the above analysis, it is clear from the last equation that the minimum of all primal lengths is less than $\frac{\Sigma_i y_i}{\alpha}$. Here we try to achieve parallelization by choosing all primal

variables whose length is less than or equal to this quantity in a phase, and increment their values till this condition is no more satisfied. At this point we end the phase and decrement the value of α.

$x \leftarrow \epsilon$
$\alpha \leftarrow n.OPT$
$y \leftarrow 0$
Algorithm NVU()
    while ($\frac{\Sigma_i y_i}{\alpha} \leq (1+\epsilon)$) {
        $(\text{length})_j = \frac{\Sigma_i a_{ij} y_i}{c_j}$
        while([S = {j : $(\text{length})_j \leq \frac{\Sigma_i y_i}{\alpha}$}] $\neq \phi$) {
            $\text{col}_{new} = \sum_{j \in S} A_j x_j$
            $\phi = \max_{\forall i} \frac{(col_{new})_i}{b_i}$
            call Round_Update()
        }
        $\alpha = \frac{\alpha}{1+\epsilon}$
    }

Round_Update()
    $x_j = x_j(1+\frac{\epsilon}{\phi}), \quad \forall j \in S$
    $\lambda_i = \Sigma_j a_{ij} x_j, \quad \forall i$
    $y_i = e^{\epsilon \lambda_i \phi}$
    $length_j = \frac{\Sigma_i a_{ij} y_i}{c_j}, \quad \forall j$

Figure 8: NVU Algorithm

Working on the lines of Bartal [1] we have proposed a multiplicative increment $(1+\frac{\epsilon}{\phi})$ in the primal values as against additive increment proposed by Luby & Nissan [4] and Kuhn [3]. Further, the increments in the primal variables are done by composing the new primal column as a weighted sum of the chosen primal columns and choosing the incrementing factor ϕ as the maximum ratio of b_i and $(col_{new})_i$.

Recall from Garg & Konemann [2] that after choosing the primal variable whose length is minimum we choose the maximum capacity edge (dual variable) which gets incremented exactly by a factor of $(1+\epsilon)$. In our approach when we choose many primal variables, it is possible that the maximum capacity edge for several primal variables may overlap. Thus, if we construct a new column as a linear combination of the primal

columns chosen we are able to exhibit the property that if for several primal columns maximum capacity edge is the same then in this new column for this edge the property is strengthened. Also, intuitively the primal variable having less length should be having high value - so while constructing the new column more weightage should be given to the primal columns whose length is less and this is covered by the weights we have chosen.

The algorithm is based upon the following claims:

4.2.1 *Claim 1:* Initially, $\alpha \leq n.OPT$

For each row (i^{th} row) choose the first variable from [1...n], which is already not covered and for which the coefficient in this row is non-zero. Let this be the j^{th} variable, then,

$$x_j = b_i / A_{ij}$$

Assign each remaining primal variable to 0. At the end of this process,

$$\alpha = \Sigma_j c_j x_j$$

In other words, we are taking the corner points of the polytrope described by the input constraint matrix such that each corner point lies on an axis. The value of α calculated this way is certainly more than the optimum value, as the value of each constraint(i) in the primal problem is atleast b_i. Also, value of each $c_j x_j \leq OPT$. Thus,

$$OPT \leq \alpha \leq n.OPT$$

4.2.2 *Claim 2:* The Algorithm takes $O(\log n/\epsilon)$ phases

At the end of each phase we decrement the value of α by a factor of $(1+\epsilon)$. As α is no more than n times the value of the optimal solution,

$$\text{no. of phases} \leq \log_{(1+\epsilon)} n$$

Thus, the number of phases is atmost $\log_{(1+\epsilon)} n$, i.e., $O(\log n/\epsilon)$.

Recall from Bartal's Algorithm [1] that primal variables having lengths in the slot $(\frac{1}{1+\epsilon})$ and 1 were chosen in a phase and the phase ended when all the lengths had crossed this slot. In Luby & Nissan [4] also, a similar approach was followed. We notice in our algorithm that this criteria of slot of $(1+\epsilon)$ length has been relaxed to cover all primal variables whose length satisfy the threshold criteria. Using this threshold criteria it is easy to see that the primal variables that are chosen are no more restricted to the $(1+\epsilon)$ slot, because the slot has expanded. This is due to the fact that during the beginning of a phase the slot is of length exactly $(1+\epsilon)$ but as the phase proceeds this threshold increases to cover more primal variables that may not be initially chosen

at the starting of the phase. Thus, we can claim that in our approach more number of primal variables are chosen during each phase and thus the end criteria is satisfied in less number of rounds.

5 Conclusion

We studied the problem of getting the optimal primal and dual solutions to positive linear programs in a parallel environment. We gave an algorithm which obtains a $(1+\epsilon)$ approximation ratio in a polylogarithmic number of rounds. Though we have not been able to give a bound on the number of iterations within a phase, we believe that the number of iterations would not exceed a polylogarithmic function in n as the algorithm follows techniques suggested by Luby, Nissan [4] and Bartal [1]. Yet, many theoretical and practical questions remain open.

One obvious question is whether the running time can be improved? In his paper, Neil Young [6] has posed an open problem of finding a parallel algorithm for solving PLPP whose running time has ϵ^{-2} instead of ϵ^{-4}. Another interesting theoretical question is the scope of this algorithm such as whether it can be applied to non-negative LPs. Finding fast sequential approximation algorithms for general linear programs could be a start in this direction.

References

[1] Yair Bartal, John W. Byers, and Danny Raz. Global optimization using local information with applications to flow control. In *IEEE Symposium on Foundations of Computer Science*, pages 303–312.

[2] Naveen Garg and Jochen Konemann. Faster and simpler algorithms for multi-commodity flows and other fractional packing problems. In *IEEE Symposium on Foundations of Computer Science*, pages 300–309.

[3] Fabian Kuhn and Roger Wattenhofer. Constant-time distributed dominating set approximation. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 25–32, New York, NY, USA, 2003. ACM Press.

[4] Michael Luby and Noam Nisan. A parallel appximation algorithm for positive linear programming. In *ACM Symposium on Theory of Computing*, pages 448–457, 1993.

[5] Sridhar Rajagopalan and Vijay V. Vazirani. Primal-dual rnc approximation algorithms for set cover and covering integer programs. 28(2):525–540, 1999.

[6] Neal Young. Sequential and parallel algorithms for mixed packing and covering. In *FOCS '01: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, page 538, Washington, DC, USA, 2001. IEEE Computer Society.

APPENDIX

We had tried two techniques for giving priorities to certain columns and rows using Thresholding and Ranking based techniques. We present herewith the source code for the two techniques.

```
#include iostream.h
#include fstream.h
#include string.h
#include math.h

float A[2][2] = {{1.0,1.0}, {2.0,1.0}}, b[ ] = {8.0,10.0}, c[ ] = {4.0,3.0};
float epsilon, delta, OF = 0.0, y[3], x[4], D, threshold;
int n = 2, m = 2, check, yyyyu;
int count[4] = {0,0,0,0};

void update_D()
{
  D = 0.0;
  for (int i = 0;i < m;i++)
    D += b[i]*y[i];
}

int main()
{
  int i,j, numr = 0, check;
  epsilon = 0.013;
  delta = (1.0 + epsilon)*pow((1.0 + epsilon)*m, -1.0/epsilon);
  threshold = 1.0*delta*m/4.0*10.0;
  for(i = 0;i < m;i++)
    y[i] = delta/b[i];
  for(i = 0;i < n;i++)
    x[i] = 0;
  update_D();
  while(D <= 1.0)
  {
    int q[n];
    float min_length;
    int p;
    int k = -1;
    for(j = 0;j< n;j++)
    {
```

```
    float temp = 0.0;
    for(i = 0;i < m;i++)
      temp += A[i][j]*y[i]/c[j];
    cout<<j<<" "<<temp<<" "<<threshold<<endl;
    if(temp < threshold)
      q[++k] = j;
  }
  cout<<k<<endl;
  int yy = k + 1;
  if(k==-1) yyyyu++;
  ++count[k];
  for(j = k;j>=0;j- -)
  {
    min_length = 1000000.0;
    for(i = 0;i < m; i++)
      if(b[i]/A[i][q[j]] < min_length)
      {
        min_length = b[i]/A[i][q[j]];
        p = i;
      }
    x[q[j]] += (b[p]/A[p][q[j]]);
    OF += c[q[j]]*(b[p]/A[p][q[j]]);
    for(i = 0;i < m;i++)
      y[i] *= (1 + (epsilon)*((b[p]/A[p][q[j]])/(b[i]/A[i][q[j]])));
  }
  for(i = 0;i < m;i++)
    cout<<y[i]<<" ";
  cout<<endl;
  if(yy != 0)
    threshold *= (1 + 1.25*epsilon);
  else
    threshold *= (1 + 1.25*epsilon);
  update_D();
  ++numr;
}
float as = log ((1 + epsilon)/delta)/log(1 + epsilon);
cout<<endl<<x[0]/as<<" "<<x[1]/as<<" "<<x[2]/as<<" "<<x[3]/as<<endl;
OF /= log ((1 + epsilon)/delta)/log(1 + epsilon);
cout<<"OF = "<<OF<<" number of rounds = "<<numr<<endl;
cout<<"Expected rounds = "<<(m/epsilon)*(log(m)/log(1 + epsilon))<<endl;
cout<<count[0]<<" "<<count[1]<<" "<<count[2]<<" "<<count[3]<<endl;
```

```
  cout<<yyyyu<<endl;
}
```

The code given above is based on a modified ranking plus thresholding system for parallelising the Garg & Konemann [2] algorithm. Since the algorithm is based on problem specific constants like the constraint vector and the constraint matrix entries, hence calculating a threshold globally and independent of the input problem is not straighforward. We used the above code to determine the constant by solving several PLPPs using it but a single threshold factor could not be determined.