Parallel algorithm for 0/1 knapsack problem

INDEX

I. Introduction

- O-1 Knapsack problem has many real world applications and is one of the popular combinatorial optimization problems. There are many algorithms available for this problem but traditionally the Dynamic programming solution is used to solve this problem.

- This paper discusses the parallel implementation of a Dynamic Programming solution on an n-cube hypercube.
- After the initial explanation of the implementation of the algorithm, authors proceed to theoretically derive the run time for the same.

II a. Knapsack problem

- **KNAP(G, c)**
- Given a set G of m objects each having a weight of wi, profit pi, and a knapsack with a maximum capacity of c

i =1

Σ *pizi is maximized*

m

subject to

i =1

Σ *wizi* $\leq$ *c*

m

where $z_i =$ *(1 - if object i is included*

(0 - otherwise

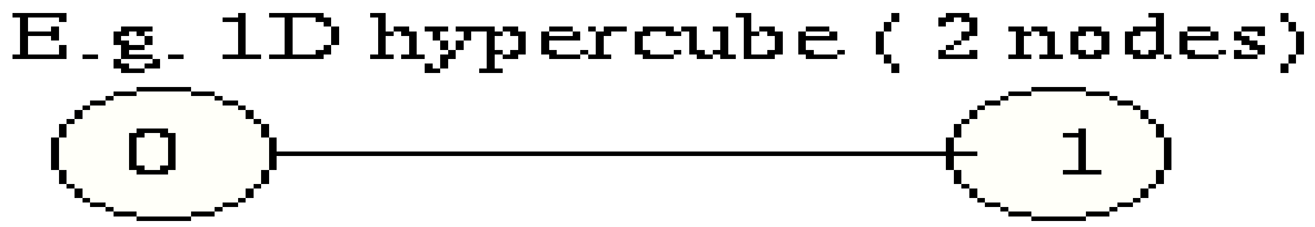
IIb. Hypercube

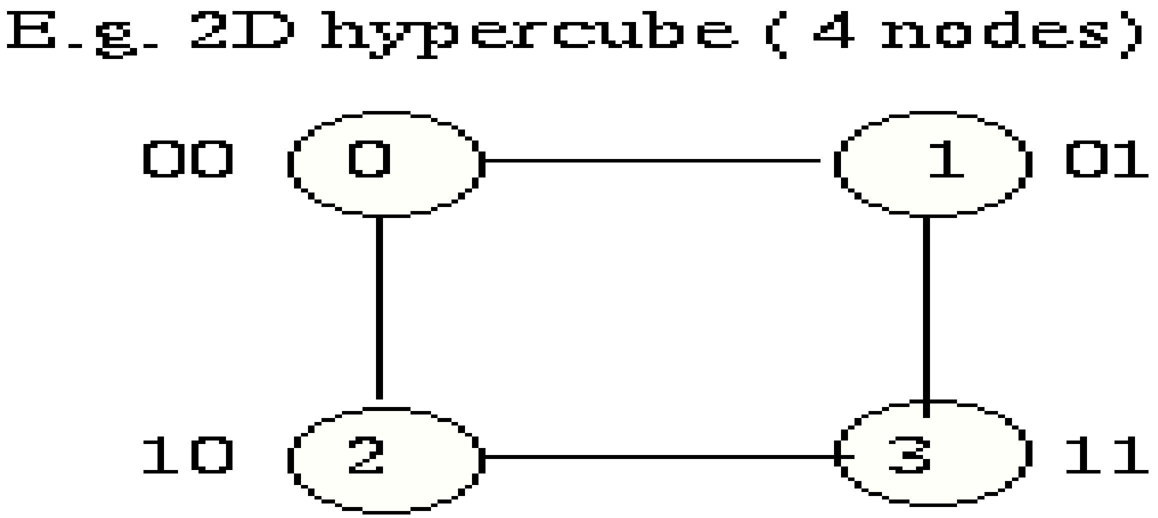
- A Hypercube or a binary n-cube contains N = 2^n nodes arranged along the n dimensions of the hypercube
- The labeling is done such that all the neighboring nodes have their labels differing in one bit position only

- Nodes that are reachable by traveling two edges differ in two bit positions
- The total number of bit positions at which two labels differ is called the ‘hamming distance’ and represents the distance between the nodes in terms of the number of edges between the nodes

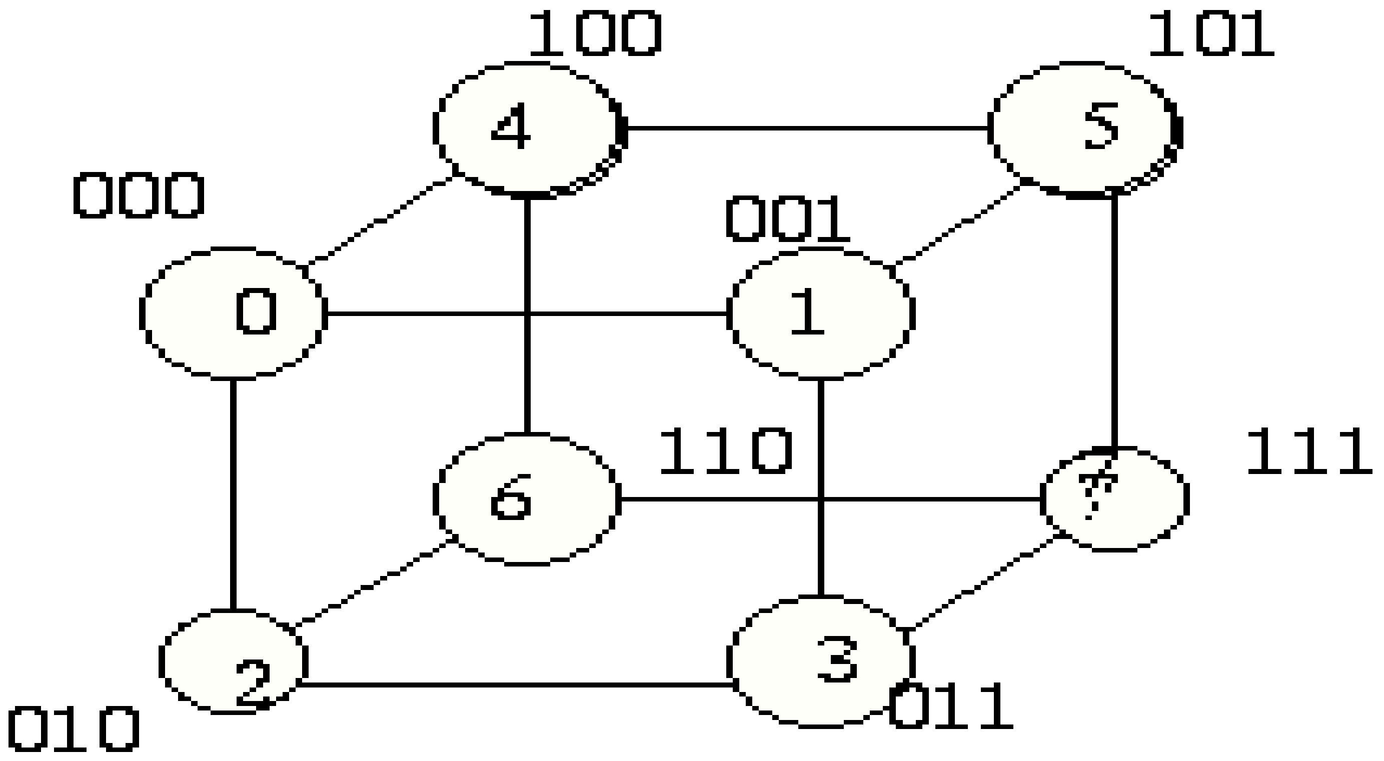
- In an n-cube hypercube, each node has n neighbors and the length of the shortest path between any two nodes is at most n.
- An n-cube hypercube can be constructed by combining two n-1 cubes and connecting the corresponding nodes

- Hypercubes are implemented in computers such as iPSC, ncube and CM-2

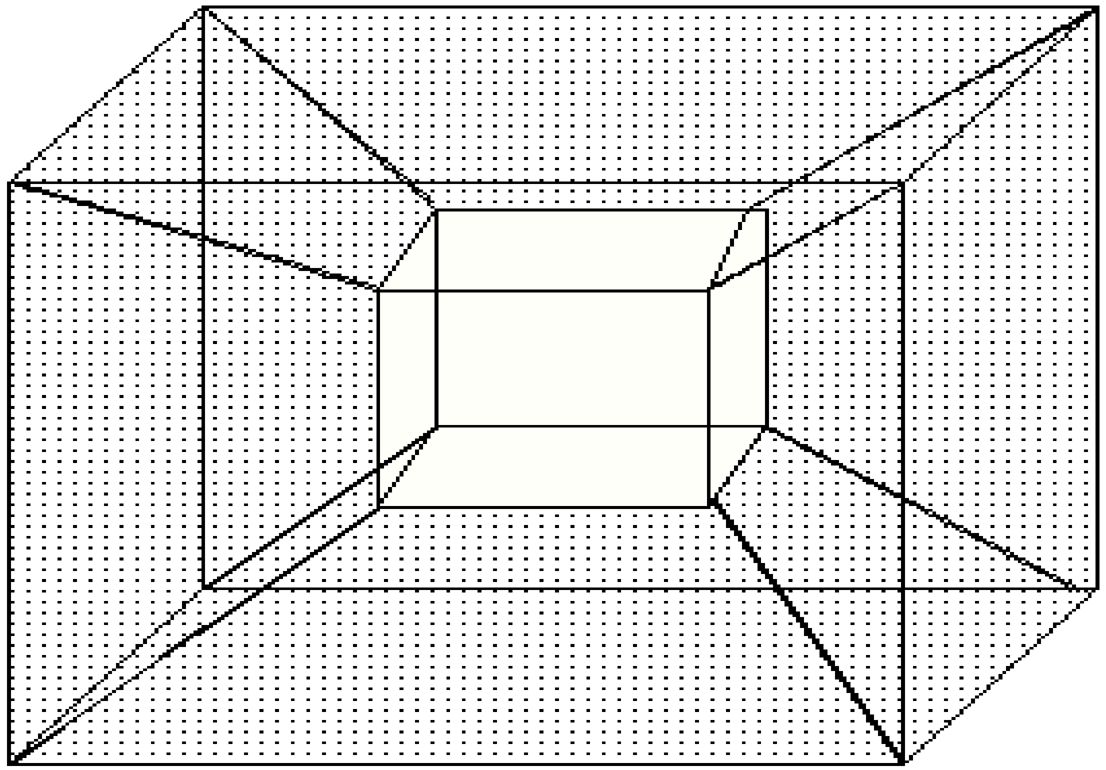

E.g. 1D hypercube (2 nodes)
0
1


E.g. 2D hypercube (4 nodes)
00
0
1
01
10
2
3
11

E.g 3D hypercube (8 nodes)



4D Hypercube or Binary 4 – Cube



III. Dynamic Programming solution for 0/1 Knapsack problem

- ***i. Optimal substructure of the problem***
- ***ii. Notations***
- ***iii. Algorithms***

i. Optimal substructure of the problem

- Let W be the weight of the objects selected in the optimal solution and object j with weight wj and profit pj be part of the optimal solution
- If ‘j’ were to be removed from the optimal set, then the optimal set without object ‘j’ would be the solution of the **KNAP(G-j, c-wj)**.

- Hence the knapsack problem has the optimal substructure and dynamic programming can be applied to solve this problem.

ii. Notations

- fk(x) is defined as the optimal value of the profit of the solution of the knapsack problem with capacity of x and with first ‘k’ items of the input set.
- fm(c) indicates the value of the optimal solution.

- According to the optimal sub structure of DP solution, we have,

f0(x) = (0 – if $x \geq 0$;

= (-∞ - if $x < 0$;

fi(x) = max{fi-1(x), fi-1(x-wi) + pi },for all x

where i = 1,2,3 ….., m

- The DP solution of this problem given below is based on these equations.
- This solution uses a data structure Si, which is a list of tuples representing the step points of fi(x).
- The solution generates ‘m’ such lists and each list represents the solution to the ‘j’th sub problem fj(x) where x = 1, 2, 3, ……c.

- The algorithm generates the solution vector ’z’ by tracing back the history by analyzing the lists
- An example of list Sj, is given below

x	0	1	2	....	c
$f_j(x)$	$f_j(0)$	$f_j(1)$	$f_j(2)$	....	$f_j(c)$

iii. Algorithms

Algorithm 1 [forward part of dynamic programming]

```
$S_0 \leftarrow \{(0, 0)\}$
for $i \leftarrow 1$ to $m$ do
begin
    $S_i' \leftarrow \{(P+p_i, W+w_i) \mid (P, W) \in S_{i-1}, W+w_i \leq c\}$
    $S_i \leftarrow merge\,(S_{i-1}, S_i')$
end
```

Algorithm 2 [backtracking part of dynamic programming]

```
    $(P, W) \leftarrow$ last tuple in $S_m$
    for $i \leftarrow m$ downto 1 do
    begin
        if $(P - p_i, W - w_i) \in S_{i-1}$ then
            $z_i \leftarrow 1$;   $P \leftarrow P - p_i$;   $W \leftarrow W - w_i$

      else
          $z_i \leftarrow 0$
      endif
end
```

- During the merge process, if (S'i U Si-1) contains two tuples (Pj, Wj) and (Pk, Wk) such that Pj ≤ Pk and Wj ≥ Wk, then (Pj, Wj) is discarded.

- The backtracking algorithm demonstrates the construction of the solution vector Z from the lists generated for all objects by the ‘forward part’ of the algorithm.
- Here for every object, it is verified to see if that object was added to the optimal set by verified the list obtained from previous iteration i.e. Si-1

IV. Parallel algorithm for 0/1 Knap sack problem

- The parallel algorithm for knapsack problem is based on the operations ‘combine’ and ‘history’

$$\mathbf{e} = combine(\mathbf{b}, \mathbf{d}),$$

$$\text{where } e_i = \max_{0 \le j \le i} \{b_j + d_{i-j}\}, \text{ for } i = 0, 1, \ldots, c.$$

$$\mathbf{h} = history(\mathbf{b}, \mathbf{d}),$$

where $h_i = j_0$ such that

$$b_{j_0} + d_{i-j_0} = \max_{0 \le i \le i} \{b_j + d_{i-j}\}, \text{ for } i = 0, 1, \ldots, c$$

- Whereas ‘combine’ operation is used to combine the solutions of two sub problems, the history operation helps in combining the history, which is required for constructing the solution vector z

- In other words, if b and d are two optimal profit vectors for **KNAP(B, c)** and **KNAP(D, c)** such that B ∩ D = Φ then e, which holds the result of the combine operation above is the optimal profit vector for **KNAP(B U D, c).**
- The proof for this is given below:

Let **z** be an optimal solution vector for $KNAP(B \cup D, x)$ for some x, where $0 \le x \le c$. Let α be a value of the optimal solution and β be the sum of weights of all selected objects. Formally, this can be stated as follows:

$$\sum_{i \in B \cup D} p_i z_i = \alpha, \quad \sum_{i \in B \cup D} w_i z_i = \beta \le x$$

Each of these expressions can be broken into two expressions as follows:

$$\sum_{i\in B} p_i z_i = \alpha_B, \quad \sum_{i\in B} w_i z_i = \beta_B,$$

and

$$\sum_{i\in D} p_i z_i = \alpha_D, \quad \sum_{i\in D} w_i z_i = \beta_D.$$

Then

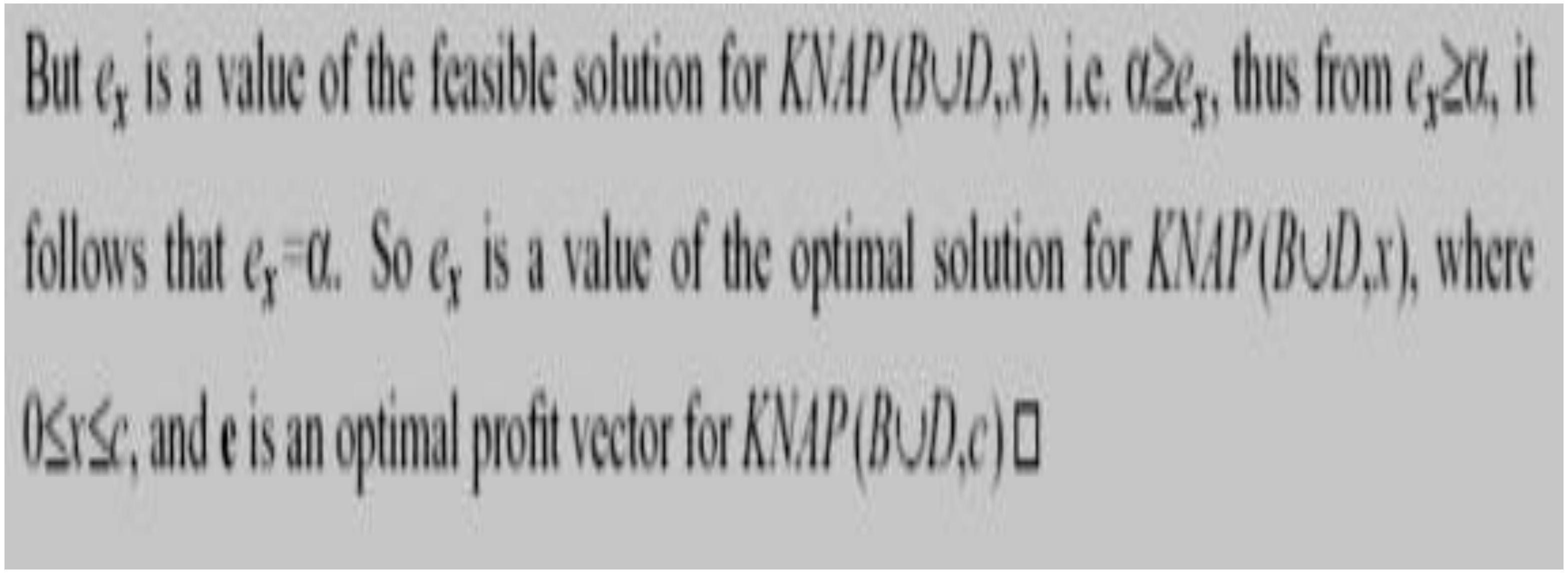
$$\alpha_B + \alpha_D = \alpha, \quad \beta_B + \beta_D = \beta.$$

Since b_{β_B} is a value of the optimal solution for $KNAP(B, \beta_B)$, then $\alpha_B \leq b_{\beta_B}$. The same is true for $KNAP(D, \beta_D)$, thus $\alpha_D \leq b_{\beta_D}$.

Thus,

$$e_x \geq e_\beta = \max_{0 \leq j \leq \beta}(b_j + d_{\beta - j}) \geq b_{\beta_B} + d_{\beta - \beta_B}$$

$$= b_{\beta_B} + d_{\beta_D} \geq \alpha_B + \alpha_D = \alpha$$



But e_x is a value of the feasible solution for $KNAP(B\cup D,x)$, i.e. $\alpha\ge e_x$, thus from $e_x\ge\alpha$, it follows that $e_x=\alpha$. So e_x is a value of the optimal solution for $KNAP(B\cup D,x)$, where $0\le x\le c$, and **e** is an optimal profit vector for $KNAP(B\cup D,c)$ □

The common approach in parallel programming is to

- *A.Divide the original problem into sub problems of smaller size.*

  It is critical in this stage to identify the parts of the problem that can be executed in parallel.
- *B.Solve the smaller sub problems in parallel*

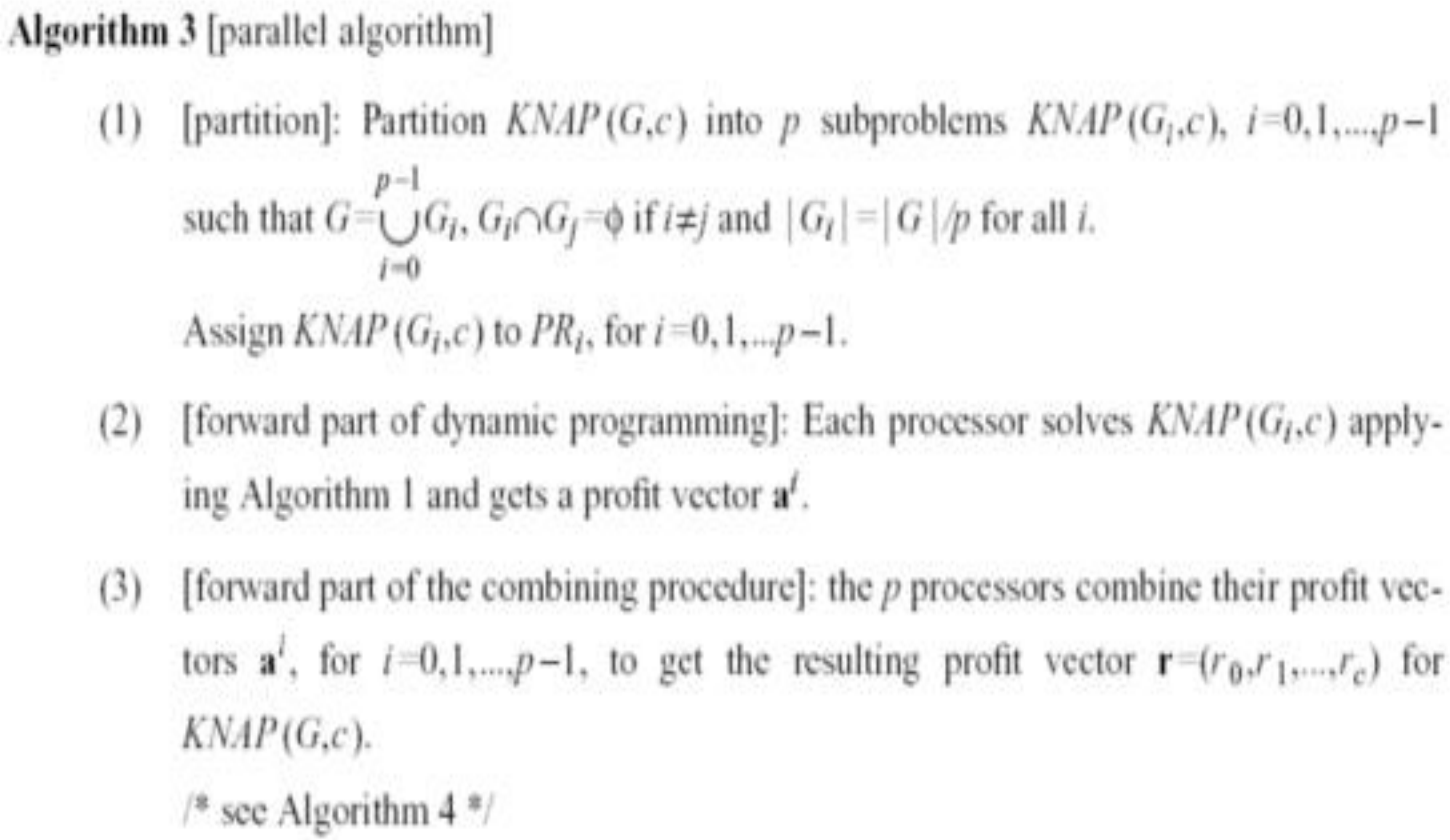
  At this stage the main objective is to lower the communication between the processors working on their respective sub problems.

- *C.Combine the solutions to the sub problems to obtain the solution to the original problem*

- If the combined overhead associated with any of the above 3 stages is more than the efficiency gained from the parallel processing, then it is not beneficial to perform parallel processing.

- However, from the above lemma we know that there is way to combine the solutions of any two sub problems.
- This also enables us to divide main problem into many sub problems, which can be executed parallel with each other and can be combined later using the combine operations.

- The algorithms that are based on the ‘combine’ and ‘history’ operations are given below:



Algorithm 3 [parallel algorithm]

(1) [partition]: Partition $KNAP(G,c)$ into p subproblems $KNAP(G_i,c)$, $i=0,1,\ldots,p-1$ such that $G=\bigcup_{i=0}^{p-1} G_i$, $G_i \cap G_j = \phi$ if $i \neq j$ and $|G_i| = |G|/p$ for all i.

Assign $KNAP(G_i,c)$ to PR_i, for $i=0,1,\ldots p-1$.

(2) [forward part of dynamic programming]: Each processor solves $KNAP(G_i,c)$ applying Algorithm 1 and gets a profit vector $\mathbf{a}^i$.

(3) [forward part of the combining procedure]: the p processors combine their profit vectors $\mathbf{a}^i$, for $i=0,1,\ldots,p-1$, to get the resulting profit vector $\mathbf{r}=(r_0,r_1,\ldots,r_c)$ for $KNAP(G,c)$.

/* see Algorithm 4 */

(4) [backtracking part of the combining procedure]: the p processors trace back the combining history to get x_l, for $l=0,1,\ldots,p-1$ such that

$$\sum_{i=0}^{p-1} x_i = c, \quad \sum_{i=0}^{p-1} a_{x_i}^{l} = r_c$$

/* see Algorithm 5 */

(5) [backtracking part of dynamic programming]: Each processor traces back its dynamic programming history applying Algorithm 2 with $(P,W)=(a_{x_l}^{l}, x_l)$ to get an optimal solution vector.

- In step 1, the algorithm will divide the original input set into p different sets such that no object is common between these subsets.
- Each of these subsets is assigned to a processor, which will perform the forward part of the single processor dynamic programming algorithm (algorithm 1).
- Even though the input size of the sub problem is reduced, all the processors will try to find solution for the same size of knapsack as that of the original problem (i.e.) c.

Algorithm 4 [forward part of the combining procedure]

```
    $l \leftarrow \log_2 c - 1$
    $\mathbf{a}^{(i,0)} \leftarrow \mathbf{a}^i$, for $i = 0, 1, \ldots p-1$
    for $k \leftarrow 1$ to $n$ do

  begin
      $g \leftarrow min(k, l)$ ; $r \leftarrow \frac{p}{2^g}$ ;
      Partition the set of $p$ processors into $r$ groups of size $2^g$ ;
      for each group $i$ $(0 \le i \le r-1)$ in parallel do
      begin
          All processors in the group $i$ compute
          $\mathbf{a}^{(i,k)} \leftarrow combine(\mathbf{a}^{(2i,k-1)}, \mathbf{a}^{(2i+1,k-1)})$
          $\mathbf{h}^{(i,k)} \leftarrow history(\mathbf{a}^{(2i,k-1)}, \mathbf{a}^{(2i+1,k-1)})$
      end
  end
```

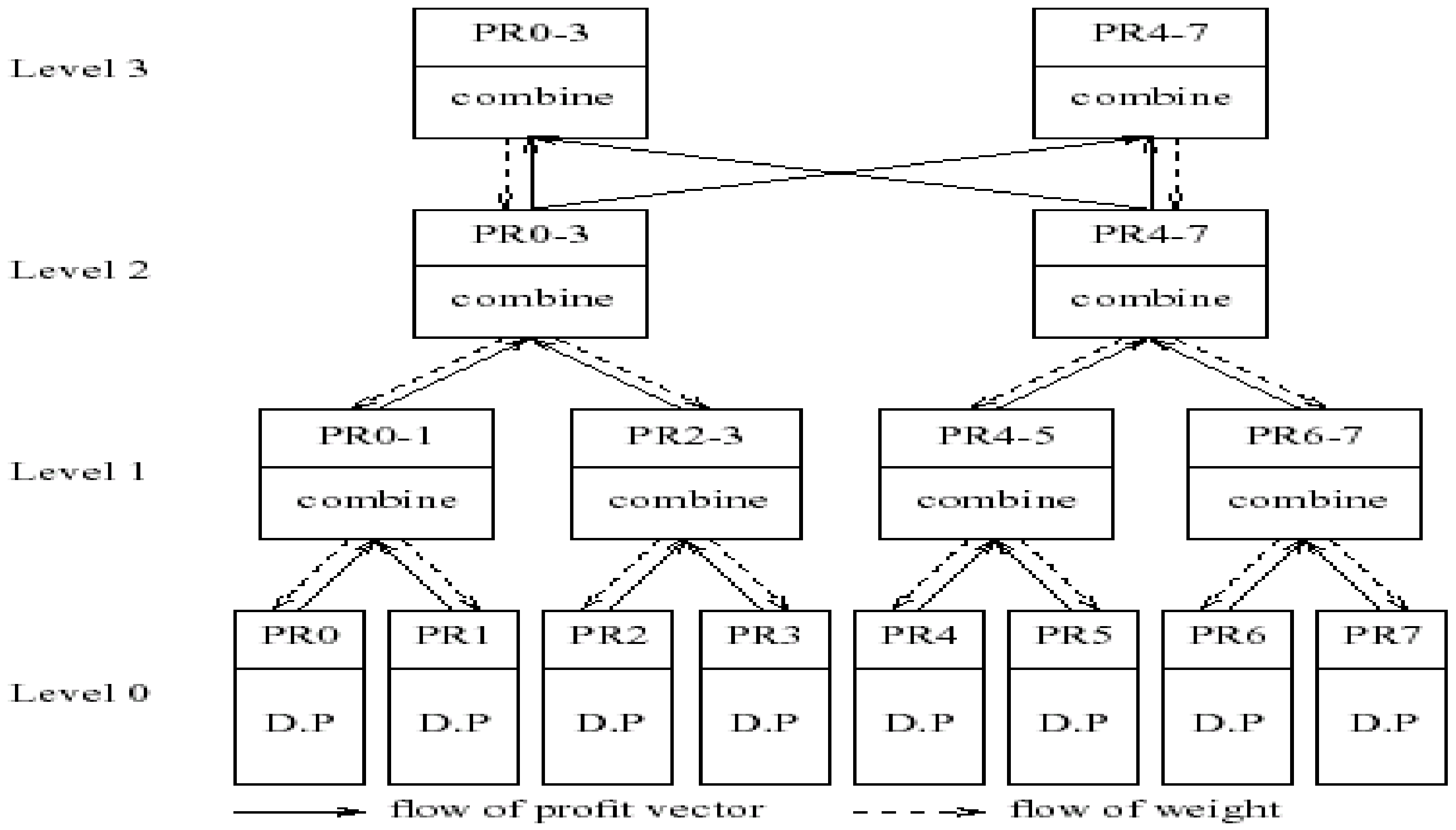
Notations used in the above algorithms

- *l – the level from which the group size does not increase.*
- *a(i, k) is the profit vector for group i at level k*
- *h(i, k) is the history vector for group i at level k*

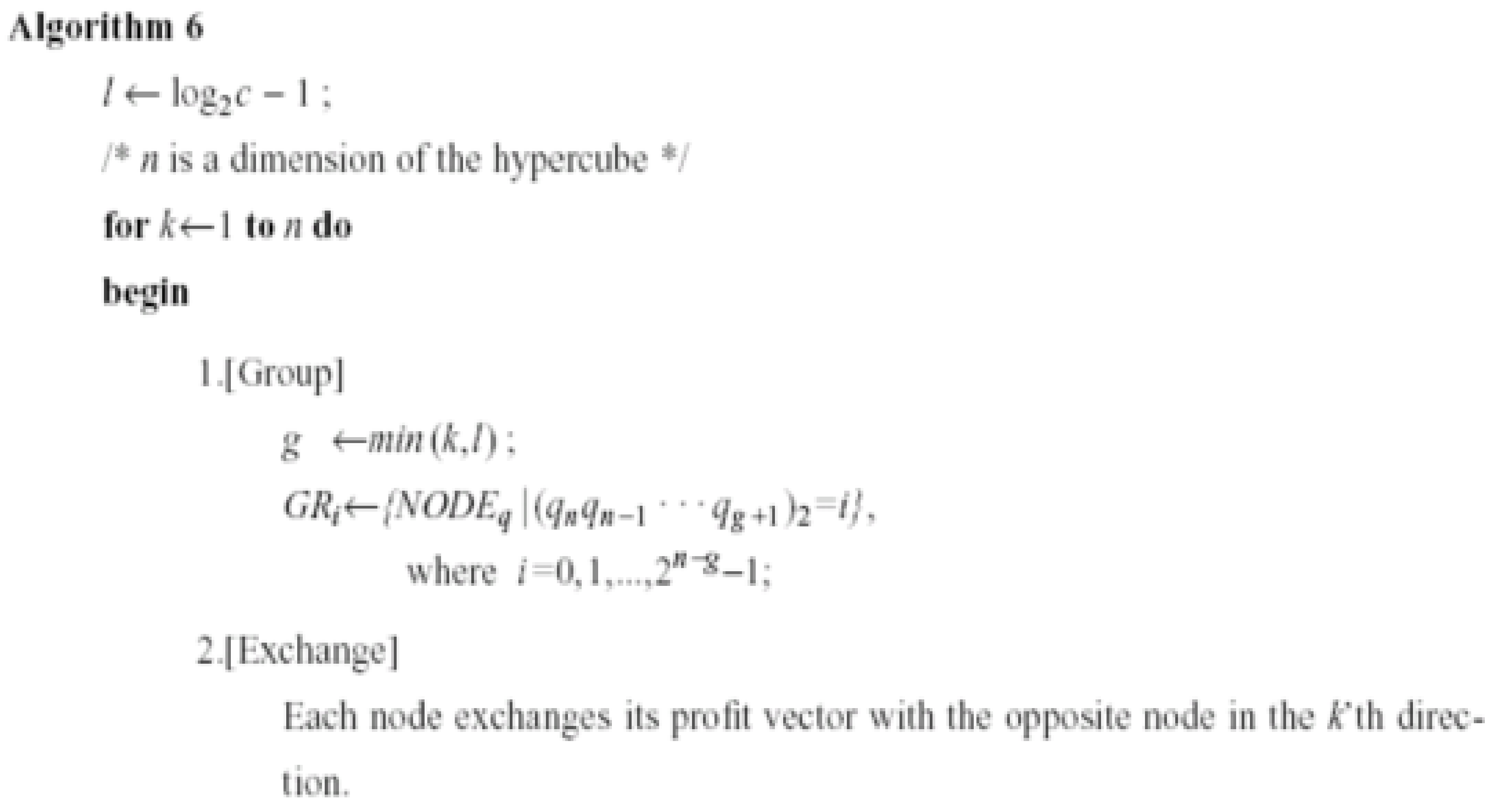
- At each level of combining procedure, the number of groups of processors is reduced to half.
- Combining procedure is applied till the final solution is obtained by combining the solutions of all the sub problems.

- Initially at level 1, there are p processors.
- The combine processes is applied to the output of these processors and for the next level we have p/2 groups of processors, each group containing 2 processors.
- This process is continued and for the next level there are p/4 groups of processors each with 4 processors.

- This process is in the form of a tree and is shown below:


Level 3
PR0-3
combine
PR4-7
combine
Level 2
PR0-3
combine
PR4-7
combine
Level 1
PR0-1
combine
PR2-3
combine
PR4-5
combine
PR6-7
combine
Level 0
PR0
D.P
PR1
D.P
PR2
D.P
PR3
D.P
PR4
D.P
PR5
D.P
PR6
D.P
PR7
D.P
flow of profit vector
flow of weight

Implementation of algorithm on hypercube



Algorithm 6

```
$l \leftarrow \log_2 c - 1$ ;
/* $n$ is a dimension of the hypercube */
for $k \leftarrow 1$ to $n$ do
begin
    1.[Group]
        $g \leftarrow min(k,l)$ ;
        $GR_i \leftarrow \{NODE_q \mid (q_n q_{n-1} \cdots q_{g+1})_2 = i\}$,
                where $i = 0,1,\ldots,2^{n-g}-1$;
    2.[Exchange]
        Each node exchanges its profit vector with the opposite node in the $k$'th direc-
        tion.
```

3.[Compute]

Every node in the group computes its part of the resulting profit vector and saves the combining history.

4.[Gather]

Every node broadcasts its elements of the combined profit vector to every other node in the group, so that every node in the group has the same new profit vector.

end

- In step (ii), each node exchanges its profit vector with its opposite node in the kth direction.
- In step (iii) each node in the group calculates its part of the profit vector.
- In step (iv), every node broadcasts the results of its computation to all the other nodes in the group. At the end of this step, all the nodes of the group have the same whole profit vector.

Analyse Time and Speed

- A. TDP(p,m,c) = Total processing time for the dynamic programming algorithm with p processors and (m,c) is the knapsack problem size and is equal to,

$$O\left(\frac{T_{DP}(1,m,c)}{p}\right).$$

- B. TCB(p, m, c) = Processing time for the combining procedure and is equal to,

$$O(c^2 + cn)$$

- C. TDT(n, c) = Total data transmission time for a hypercube of nth order and c is the size of the knapsack and is equal to,

$$O(n^2\beta + cn\gamma).$$

Where, β is the communication setup time and γ is the unit data transfer time.

- D. THN(n, m) = Total data transmission time from host to all nodes and is equal to

$$O(2^n\beta' + m\gamma').$$

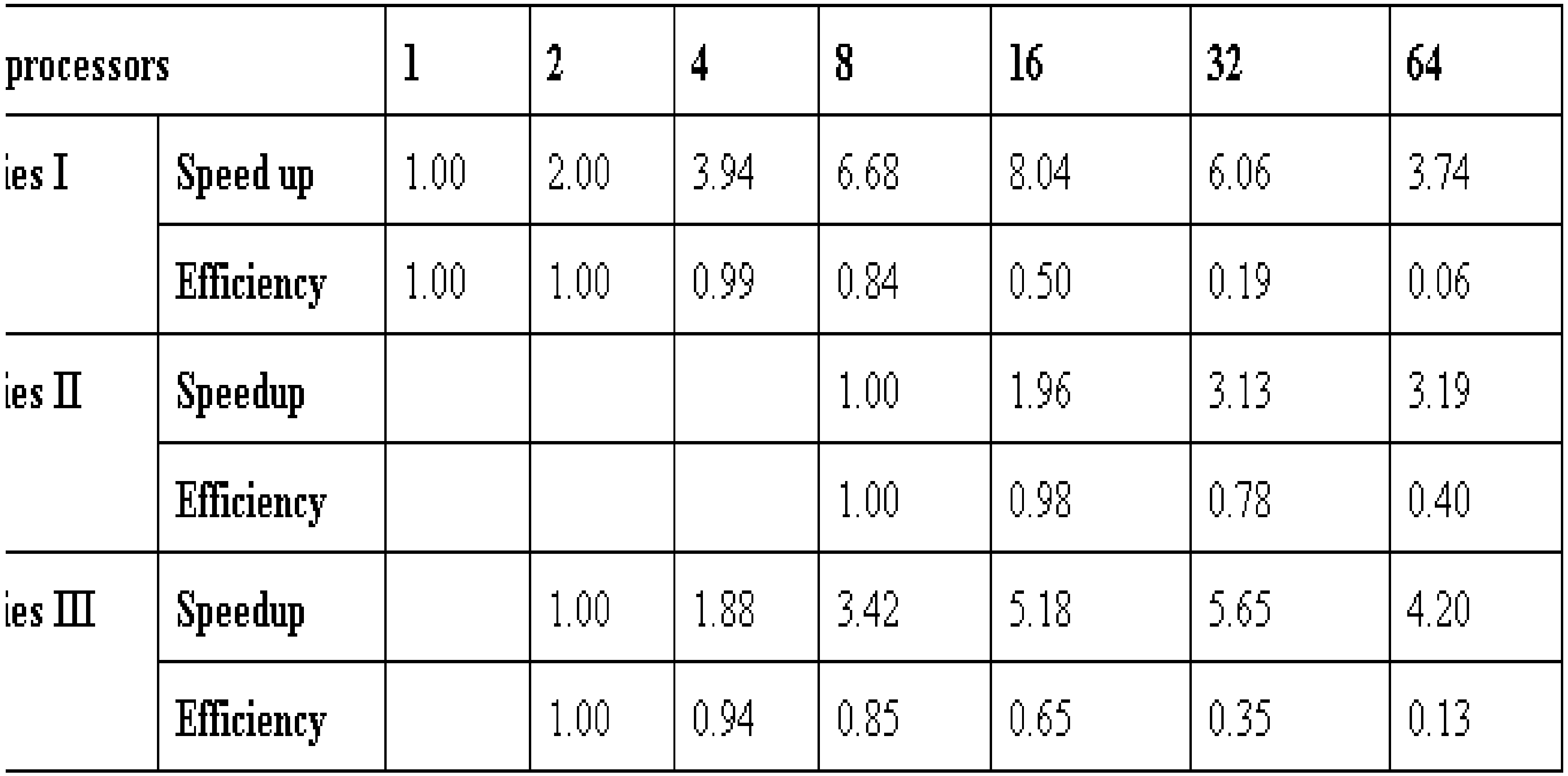
where, β' is the communication setup time between host and node and γ' is the unit data transfer between host and node.

V. Conclusions

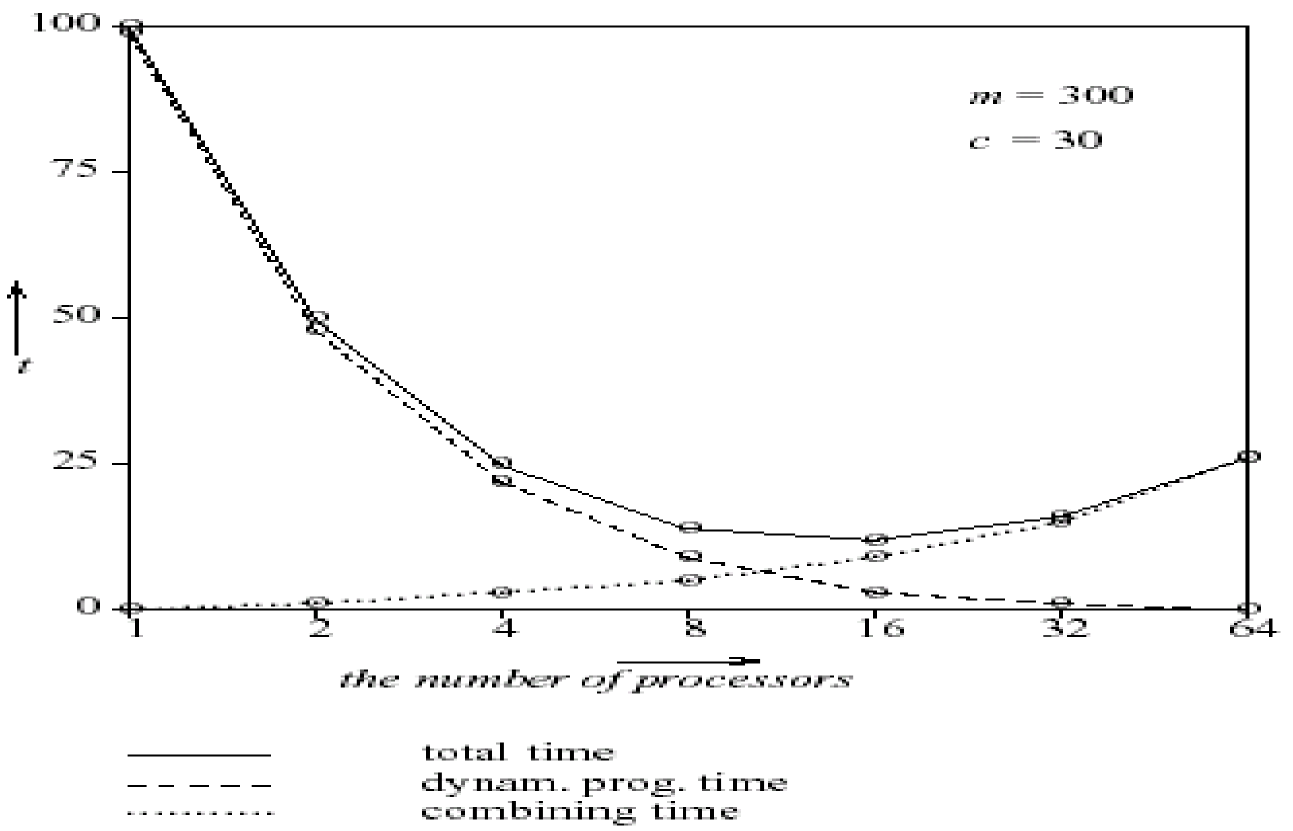
- ***Series I:*** For this experiment a knapsack problem with 300 objects and size of 30 is used and 20 instances of the problem are used. The solution was implemented on a hyper cube with number of active nodes p = 1, 2, 4, 8, 16, 32, 64.

- ***<u>Series II:</u>*** For this experiment, a knapsack problem with 1000 objects and size of 100 is used and 6 instances of the problem are tested.

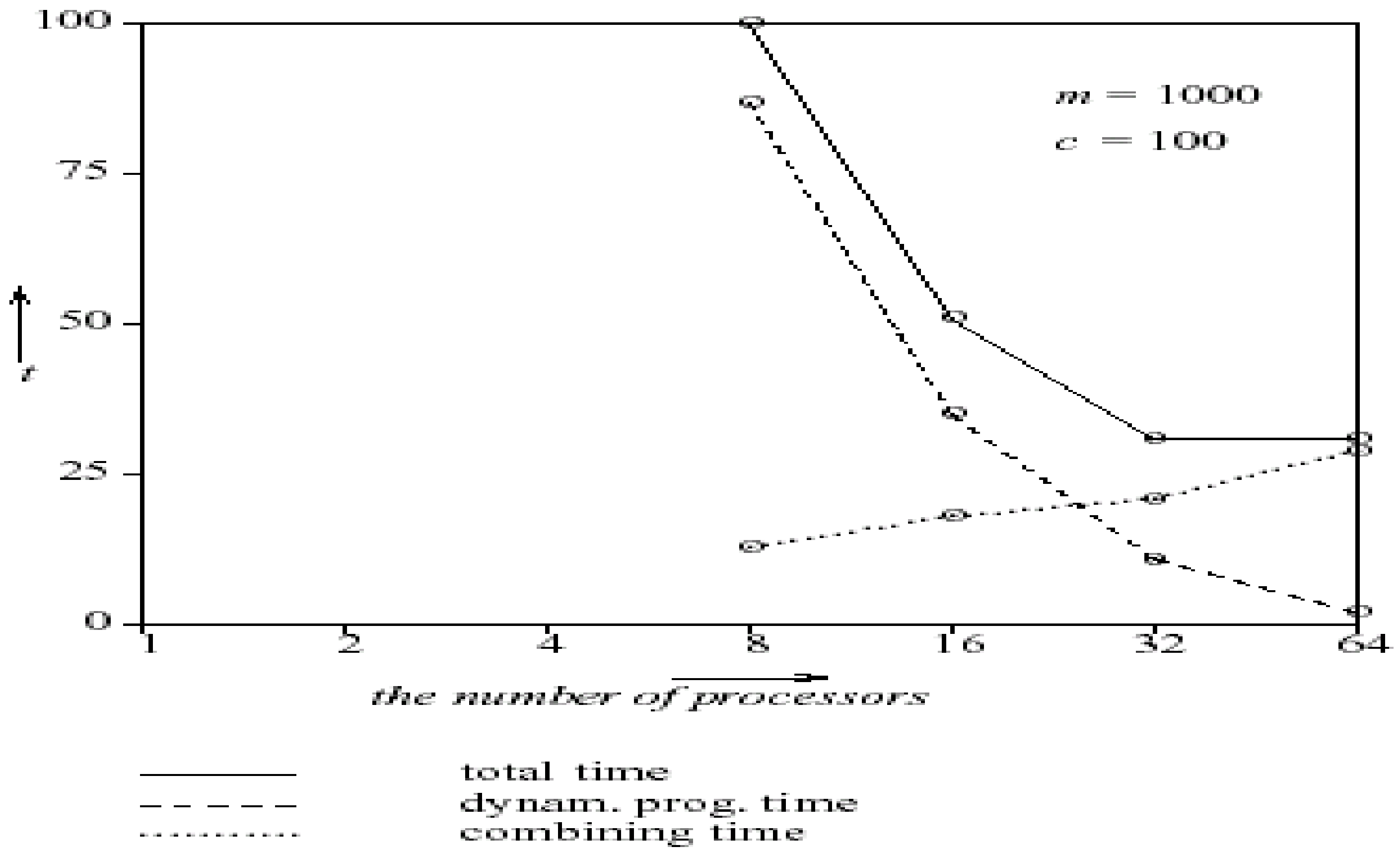
- ***Series III:*** This test was conducted for two-dimensional knapsack problem. It tested 12 instances of a problem with 500 objects and two constraints equal to 10 and 5 respectively.



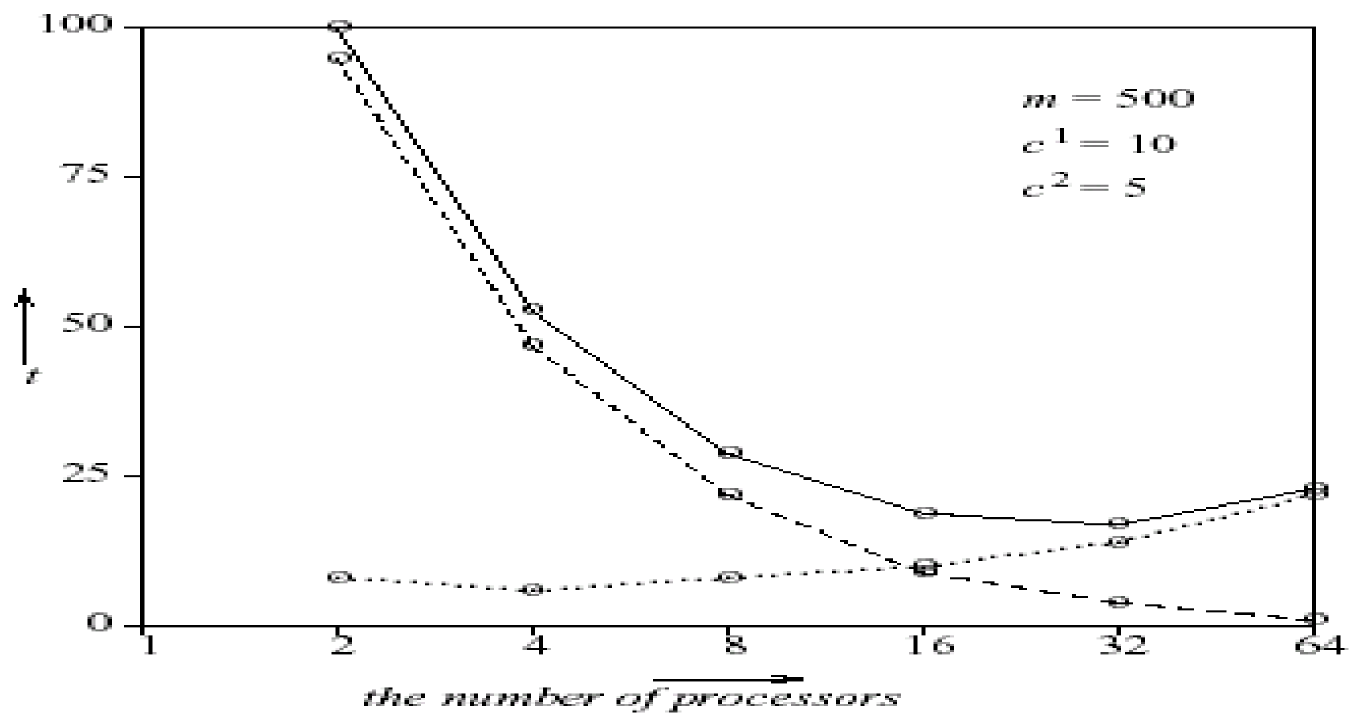
processors		1	2	4	8	16	32	64
ies I	Speed up	1.00	2.00	3.94	6.68	8.04	6.06	3.74
	Efficiency	1.00	1.00	0.99	0.84	0.50	0.19	0.06
ies II	Speedup				1.00	1.96	3.13	3.19
	Efficiency				1.00	0.98	0.78	0.40
ies III	Speedup		1.00	1.88	3.42	5.18	5.65	4.20
	Efficiency		1.00	0.94	0.85	0.65	0.35	0.13



Components of elapsed time for series-1



Components of elapsed time for series-2



———————	total time
– – – – –	dynam. prog. time
··············	combining time

Components of elapsed time for series-3

References

- A Hypercube Algorithm for the 0/1 Knapsack Problem, by Jong Lee, Eugene Shragowitz, Sartaj Sahni, University of Minnesota.
- Text Book on ‘ Fundamentals of Parallel Processing’ by Harry F. Jordan and Gita Alaghband.