Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Parallel time and space upper-bounds for the subset-sum problem

C.A.A. Sanches [a,*], N.Y. Soma [a], H.H. Yanasse [b]

[a] ITA, Brazil
[b] INPE, Brazil

ARTICLE INFO

Article history:
Received 11 October 2007
Received in revised form 17 June 2008
Accepted 27 June 2008
Communicated by J. Diaz

Keywords:
Subset-sum problem
Knapsack problem
Parallel algorithms
Dynamic programming
Upper-bound complexity

ABSTRACT

Three new parallel scalable algorithms for solving the Subset-Sum Problem in $O(\frac{n}{p}(c - w_{\min}))$ time and $O(n + c)$ space in the PRAM model are presented, where n is the number of objects, c is the capacity, $w_{\min}$ is the smallest weight and p is the number of processors. These time and space bounds are better than the direct parallelization of Bellman's algorithm, which was the most efficient known result.

1. Introduction

Given a set of n objects $A = \{a_0, a_1, \ldots, a_{n-1}\}$, each one with a weight $w_i \in Z^+$ and profit $p_i \in Z^+$, $0 \leq i < n$, the *0–1 Knapsack Problem*, named KP01(A, c) or just KP01, is to find the most profitable subset amongst objects from A without exceeding the knapsack capacity of $c \in Z^+$. Alternatively, determine a binary n-tuple $X = (x_0, x_1, \ldots, x_{n-1})$ which maximizes $\sum_{i=0}^{n-1} p_i x_i$, subject to $\sum_{i=0}^{n-1} w_i x_i \leq c$. Also, without loss of generality, admit that $\sum_{i=0}^{n-1} w_i > c > w_{\max} > w_{\min} > 0$, where $w_{\max}$ and $w_{\min}$ are, respectively, the largest and the smallest weight.

The algorithms for exactly solving KP01 can be classified into two paradigms: the force brute and the dynamic programming. The first one appears with Horowitz and Sahni [9], that uses two lists of partial solutions in a time and space bounded by $O(2^{n/2})$. Notice that this algorithm is a variation of the immediate brute force approach of $O(2^n)$, with an improvement of a square-root factor. This result still is the sequential time *upper-bound* if in the complexity analysis just the quantity of objects is considered. Within that quantity constraint – just n is to be considered – Sanches et al. [16] improved recently the parallel time *upper-bound* to a variant of the KP01, the well-known *Subset-Sum Problem* (SSP), defined when the profits are equal to the weights of the objects, that is, $p_i = w_i$, $0 \leq i < n$. The model used is a CREW PRAM *(Concurrent Read/Exclusive Write)* with p processors, where $1 \leq p \leq 2^{n/2}/n^2$, and the result obtained was that it is possible to solve the SSP in a time $O(2^{n/2}/p)$ and space $O(2^{n/2})$.

To the other approach, since the fifties of the last century, KP01 is sequentially solved in time and space bounded by $O(nc)$ by the well-known Bellman's dynamic programming paradigm [3]. Based on it, many parallel algorithms for the KP01 appeared in the literature [8,11–14,18,7]. Kindervater and Lenstra [11] and, latter on, Lin and Storer [13] suggested a direct parallelization for the Bellman's approach. To a EREW PRAM *(Exclusive Read/Exclusive Write)* with p processors, this algorithm demands time $O(nc/p)$ and space $O(nc)$. To the best of our knowledge, this is the best current parallel time and space *upper-bounds* for the KP01.

* Corresponding address: Instituto Tecnológico de Aeronáutica (ITA) - Praça Mal. Eduardo Gomes, 50 - 12228-900 - São José dos Campos - São Paulo, Brazil. Tel.: +55 12 3947 5899.
E-mail address: alonso@ita.br (C.A.A. Sanches).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.06.051

This algorithm can be simulated by a hypercube with p processors in time $O(\frac{nc}{p}\log p)$ and space $O(nc)$ [13]. Recently, Goldman and Trystram [7] introduced an algorithm also for the hypercube to the *Unbounded Knapsack Problem*, that is, when the variables are non-negative integers instead of 0 or 1, which is bounded in time by $O(\frac{nc}{p}+\frac{c}{w_{\min}})$ to a hypercube with $p<\frac{c}{\log w_{\min}}$ processors. Although these algorithms can be efficiently implemented, they do not improve the time and space *upper-bounds* for the KP01.

For the SSP, there are specific algorithms also based on dynamic programming, but different from Bellman's approach. Yanasse and Soma (cf. [10], pp. 90–91) introduced an algorithm to sequentially solve the problem in $O(n(c-2w_{\min})+c)$ time and space $O(n+c)$; Soma and Toth [17] gave a sequential algorithm which combines dynamic programming and the *two-list* enumeration to obtain a time complexity of $O((n-\log c^2)(c-2w_{\min})+c+n)$ and space $O(n+c)$. Both algorithms configure an improvement in terms of computational resources related to Bellman's approach, and, to the present, no study was carried out on their parallelizations.

Based on those two latter works, we introduce three new algorithms for solving the SSP in the PRAM model – *Algorithm* 1, *Algorithm* 2a and *Algorithm* 3 –, which improve the parallel time and space *upper-bounds*. They are the only ones known in the literature to be bounded by $O(\frac{n}{p}(c-w_{\min}))$ in time and $O(n+c)$ in space: the first two are derived from Yanasse and Soma algorithm, and the third corresponds to Soma and Toth algorithm.

A PRAM is usually considered just a theoretical model due to infeasibility of efficient and practical implementations. However, the results obtained in this model do fully express the relations that can be processed simultaneously and therefore they have a wide and longstanding impact. A series of recent new algorithms [15,19,4,6] do confirm the interest in finding new time or space *upper-bounds* for many a great variety of problems to this model.

2. Algorithm 1: For $p \le w_{\min}$ processors

The Yanasse and Soma algorithm (cf. [10], pp. 90–91) for the SSP is based on a variation of Bellman's paradigm. It uses a single vector g with $c+1$ positions, that is filled in such a way that, if there is a feasible filling of the knapsack with capacity k such that an item w_i has the largest index present in that solution, then $g[k]=i$, where $0\le k\le c$, $w_i\in Z^+$ and $0\le i<n$. Recall that $w_{\min}=\min\{w_0,\dots,w_{n-1}\}$. If a capacity k admits multiple solutions, then just the smallest index among those solutions is stored in g. Moreover, $g[k]=n$ indicates that there is no knapsack filling with capacity k.

The Yanasse and Soma algorithm has basically four phases:

(I) Initialization of g: all $c+1$ positions are set to n, this indicates that there is no feasible allocation for these knapsack capacities.
(II) Insertion of the initial solutions in g: they correspond to the allocation of a single weight w_i, $0\le i<n$.
(III) Evaluation of the remaining g values: the indices between $w_{\min}$ and $c-w_{\min}$ are evaluated in crescent order for the remaining knapsack filling.
(IV) Recovering of the optimal solution X: from g, the weights in an optimal solution filling are obtained.

This algorithm, that fills the vector g and finds the optimal solution s, is:

```
// Phase (I): initialization of vector g
for $i \leftarrow 0$ to $c$ do
    $g[i] \leftarrow n$
// Phase (II): initial solutions with a single weight $w_j$
for $i \leftarrow n-1$ downto 0 do
    $g[w_i] \leftarrow i$
// Phase (III): evaluation of vector g
$s \leftarrow w_{\max}$
for $i \leftarrow w_{\min}$ to $c-w_{\min}$ do
    for $j \leftarrow g[i]+1$ to $n-1$ do
        $w' \leftarrow i+w_j$
        if $w' \le c$ then
            $g[w'] \leftarrow \min\{g[w'], j\}$
            $s \leftarrow \max\{s, w'\}$
```

In sequence, it is given a way to recover the binary vector X in time $O(n)$:

```
// Phase (IV): recovering the optimal solution in vector X
for $i \leftarrow 0$ to $n-1$ do
    $x_i \leftarrow 0$
$i \leftarrow s$
while $i \ne 0$ do
    $x_{g[i]} \leftarrow 1$
    $i \leftarrow i - w_{g[i]}$
```

Besides its simplicity, this algorithm is more efficient than Bellman's approach, since it solves the SSP in an overall time bounded by $O(n(c - 2w_{\min}) + c)$ and only $O(n + c)$ in space.

In *Algorithm* 1, which can be executed by a CREW PRAM, the $c - 2w_{\min}$ positions of vector g are grouped in blocks of size p. At each step, a processor evaluates a single position in the current block and it tests sequentially all of the n weights, but it also considers just the g values already computed, *i.e.*, those values with smaller indices. To do this in parallel, it is necessary to allow simultaneous accesses to vector g, and also that $p \le w_{\min}$.

The algorithm is presented next. Vector g will need $d = w_{\max} - w_{\min} - 1$ extra positions in its initialization, all of them with indices smaller than 0: this is necessary to allow the accesses executed while processing the first block. Notice that variable t_i examines all the positions of a block of g associated with processor P_i.

```
**Algorithm 1 (for $p \le w_{\min}$)**
$d \leftarrow w_{\max} - w_{\min} - 1$
// Phase (I): initialization of vector g
for $k \leftarrow 0$ to $\lceil (c + d)/p \rceil - 1$ do
// k-th block of vector g with size p
    for $i \leftarrow 0$ to $p - 1$ do in parallel
        if $kp + i - d \le c$ then
                            $g[kp + i - d] \leftarrow n$
for $i \leftarrow 0$ to $p - 1$ do in parallel
    $s_i \leftarrow 0$
// Phase (II): initial solutions with a single weight $w_j$
for $k \leftarrow 0$ to $\lceil (c - w_{\min})/p \rceil - 1$ do
    for $i \leftarrow 0$ to $p - 1$ do in parallel
        // $t_i$: next position in the k-th block of g
        $t_i \leftarrow kp + i + w_{\min}$
        if $t_i \le c$ then
                for $j \leftarrow n - 1$ downto 0 do
                    if $w_j = t_i$ then
                            $g[t_i] \leftarrow j$
                            $s_i \leftarrow \max\{s_i, t_i\}$
// Phase (III): transversing and determining vector g, block to block
for $k \leftarrow 0$ to $\lceil (c - 2w_{\min})/p \rceil - 1$ do
    for $i \leftarrow 0$ to $p - 1$ do in parallel
        // $t_i$: next position in the k-th block of g
        $t_i \leftarrow kp + i + w_{\min} + 1$
        if $t_i \le c$ then
                for $j \leftarrow n - 1$ downto 0 do
                // The weight indices are transversed in decreasing order
                    if $(g[t_i] > j)$ and $(g[t_i - w_j] < j)$ then
                            $g[t_i] \leftarrow j$
                            $s_i \leftarrow \max\{s_i, t_i\}$
```

Phase (I) is bounded in time by $O(\frac{c+w_{\max}-w_{\min}}{p})$, while *Phases* (II) and (III) by $O(\frac{n}{p}(c - w_{\min}))$. Each processor P_i, $0 \le i < p$, will have its partial solution s_i, that will be the better among those it found. In final, the optimal solution of SSP will be $s = \max\{s_i\}$, $0 \le i < p$, that can be found in parallel for the p processors in time $O(\log p)$.

In *Phase* (II), initially vector g has as values $g[i] = n$, $0 \le i \le c$. For $i < w_{min}$, it is clear that $g[i] = n$, since the eventual solutions that correspond to the knapsack filling with a single item can only occur at positions $w_{\min} \le i \le c$.

The major obstacle to determine the algorithm parallelization is given in *Phase* (III), since there can exist simultaneous reading and writing. In this phase, vector g is determined obeying an increasing order of indices. It is clear that $w_{\min}$ is the first feasible solution, since $g[i] = n$ for $i < w_{\min}$. During the evaluation of $g[i]$, w_j is added to i, $0 \le j \le n$. Therefore, if $i > c - w_{\min}$, it is not possible to change the values stored into $g[i]$, and this implies that the range interval of g to this specific phase has size $c - 2w_{\min}$.

Notice that, in *Phase* (IV) the determination of vector X is inherently sequential, since x_{i+1} needs to be found before than x_i, $0 \le i < n - 1$. Moreover, it takes $O(n)$ in time, and this implies that there is no change in the overall time bound.

Therefore, the total time is $O(\frac{n}{p}(c - w_{\min}) + \frac{c+w_{\max}-w_{\min}}{p} + n + \log p) = O(\frac{n}{p}(c - w_{\min}))$, while the space is limited to $O(n + c)$.

3. Algorithm 2: For $p \le n$ processors

In this second algorithm, the parallelization of *Phase* (III) is carried out in a different way. For the sake of comprehension, it is described first the case $p = n$ processors; after this the result is then generalized.

For $p = n$, in *Phase* (III)'s loop, at a single position of vector g, each processor P_i, $0 \le i < n$, is in charge of testing weight w_i with the value of that position. This generates the problem of simultaneous writing attempts in g, which will occur if two distinct weights w_i's possess the same value. This potential conflict can be overcome by a previous sorting of weights and with a subsequent filling of an auxiliary vector *enabled* of dimension n: its i-th position will indicate whether or not the existence of another weight – with smaller index – with the same value w_i, $0 \le i < n$. With this information, every processor will have condition of deciding whether it is its turn to write in g or not, thus avoiding a conflict.

The algorithm is given next. In the call of function *ParallelSort*, W is passed by reference, *i.e.*, it returns its weights sorted in non-decreasing order.

```
**Algorithm** 2 (**for** $p = n$)
// Phase (I): initialization of vector g
for $k \leftarrow 0$ to $\lceil c/p \rceil - 1$ do
    for $i \leftarrow 0$ to $p - 1$ do in parallel
        if $kp + i \le c$ then
                    $g[kp + i] \leftarrow n$
$ParallelSort(W)$
// In vector enabled, its first position is true
$enabled[i] \leftarrow$ true, $(i = 0)$
// The remaining positions values depend on the following test
if $w_i = w_{i-1}$, $(i > 0)$
            then $enabled[i] \leftarrow$ false
            else $enabled[i] \leftarrow$ true
$s_i \leftarrow 0$
// Phase (II): initial solutions have a single weight $w_i$
for $i \leftarrow 0$ to $p - 1$ do in parallel
    if $enabled[i]$ then
                $g[w_i] \leftarrow i$
                $s_i \leftarrow \max\{s_i, w_i\}$
// Phase (III): transversing and determining g
for $j \leftarrow w_{\min}$ to $c - w_{\min}$ do
    for $i \leftarrow 0$ to $p - 1$ do in parallel
        // Test to avoid eventual writing conflicts in g
        if $(i = g[j] + 1)$ or $(i > g[j]$ and $enabled[i])$ then
                    $w' \leftarrow j + w_i$
                    if $w' \le c$ then
                            $g[w'] \leftarrow \min\{g[w'], i\}$
                            $s_i \leftarrow \max\{s_i, w'\}$
```

In this algorithm, there is no reading conflicts in vector W, since in the generation of vector *enabled* each processor consults different weights. Therefore, it can be executed by an EREW PRAM.

Phases (I), (II) and (III) spent time $O(c/n)$, $O(1)$ and $O(c - 2w_{\min})$, respectively. The time to sort in parallel n weights by using n processors is bounded by $O(\log^2 n)$ [2] (could be $O(\log n)$ [5], although there will be no impact in the final complexity determination), the identification of the optimal solution demands time $O(\log p) = O(\log n)$, and the *Phase* (IV) is bounded by $O(n)$ via the same algorithm. Therefore, the overall time for this algorithm is $O((c - 2w_{\min}) + \frac{c}{n} + n)$. The space requirement continues to be $O(n + c)$.

This algorithm is clearly scalable provided that $p \le n$. To achieve this, it is necessary that every processor P_i, $0 \le i < p$, performs the tests of $q = \lceil n/p \rceil$ weights when constructing g, more specifically, for weights w_j where $iq \le j < \min\{(i + 1)q, n\}$. Vector *enabled* will have dimension p and its i-th position, $0 \le i < p$, will be **false** whenever the first weight of P_i equals to the last weight of P_{i-1}, *i.e.*, $w_{iq} = w_{iq-1}$, and **true** otherwise.

The parallel scalable algorithm is presented next:

```
**Algorithm** 2 (**for** $p \le n$)
// Phase (I): initialization of vector g
for $k \leftarrow 0$ to $\lceil c/p \rceil - 1$ do
    for $i \leftarrow 0$ to $p - 1$ do in parallel
        if $kp + i \le c$ then
                    $g[kp + i] \leftarrow n$
$ParallelSort(W)$
// q: size of each block of vector W
$q \leftarrow \lceil n/p \rceil$
// In vector enabled, its first position is true
```

```
$enabled[i] \leftarrow$ **true**, $(i = 0)$
// The remaining positions values depend on the next test
**if** $w_{iq} = w_{iq-1}$, $(i > 0)$
            **then** $enabled[i] \leftarrow$ **false**
            **else** $enabled[i] \leftarrow$ **true**
$s_i \leftarrow 0$
// Phase (II): initial solutions consider a single weight $w_{iq+k}$
**for** $i \leftarrow 0$ **to** $p - 1$ **do in parallel**
   // Every processor have q weights of W
   **for** $k \leftarrow q - 1$ **downto** 0 **do**
      **if** $iq + k < n$ **then**
               // Tests to avoid writing conflicts in g
               **if** $(w_{iq+k} > w_{iq})$ **or** $(w_{iq+k} = w_{iq}$ **and** $enabled[i])$ **then**
                           $g[w_{iq+k}] \leftarrow iq + k$
                           $s_i \leftarrow \max\{s_i, w_{iq+k}\}$
// Phase (III): transversing and determining g
**for** $j \leftarrow w_{\min}$ **to** $c - w_{\min}$ **do**
   **for** $i \leftarrow 0$ **to** $p - 1$ **do in parallel**
      **for** $k \leftarrow 0$ **to** $q - 1$ **do**
         **if** $iq + k < n$ **then**
           // Tests to avoid writing conflicts in g
           **if** $(iq + k = g[j] + 1)$ **or** $((iq + k > g[j])$
             **and** $((w_{iq+k} > w_{iq})$ **or** $(w_{iq+k} = w_{iq}$ **and** $enabled[i])))$ **then**
               $w' \leftarrow j + w_{iq+k}$
               **if** $w' \leq c$ **then**
                     $g[w'] \leftarrow \min\{g[w'], iq + k\}$
                     $s_i \leftarrow \max\{s_i, w'\}$
```

In a similar manner as in the previous algorithm, there is no reading conflicts in W: while determining vector *enabled*, every processor uses different weights, and, to determine vector g, each one reads a single data block.

Phases (I), (II) and (III) demand, respectively, time $O(c/p)$, $O(n/p)$ and $O(\frac{n}{p}(c - 2w_{\min}))$. The parallel sorting of n weights by using $p \leq n$ processors in an EREW PRAM is bounded by $O((\frac{n}{p} + \log^2 n) \log n)$ [1], the identification of the optimal solution demands time $O(\log n)$, and the *Phase* (IV) continues to be bounded by $O(n)$. Therefore, the overall time spent by this parallel scalable algorithm is $O(\frac{n}{p}(c - 2w_{\min}) + \frac{c}{p} + \frac{n}{p} \log n + n)$. The space remains $O(n + c)$.

Let the time complexity be considered into the following intervals: (a) $p \geq \log_2 n$ and (b) $p \leq \log_2 n$. In the first case, the time complexity is $O(\frac{n}{p}(c - 2w_{\min}) + \frac{c}{p} + n) = O(\frac{n}{p}(c - 2w_{\min}))$; in the second case, the time complexity is $O(\frac{n}{p}(c - 2w_{\min} + \log n) + \frac{c}{p})$. For convenience, the algorithm will be referred as *Algorithm* 2a and *Algorithm* 2b for cases (a) and (b), respectively.

4. Algorithm 3: $\log_2(n - 2\log_2 c) \leq p \leq n - 2\log_2 c$ processors

Soma and Toth [17] conceived a algorithm for the SSP, which combines the dynamic programming and the *two-list* paradigms. In short, it has four main phases:

(1) Among the n objects of the knapsack, choose the first $\log_2 c$ ones and determine by exhaustive enumeration all the $2^{\log_2 c} = c$ possible combinations of those weights, storing them in a vector g_1 as in Yanasse and Soma algorithm, *i.e.*, $g_1[k] = i$ means that there is a filling of the knapsack with capacity k such that the largest index of the weights in this partial solution is i. If two or more filling possess the same capacity, g_1 stores the smallest index among the largest ones present in those solutions.
(2) For other $\log_2 c$ objects, generate a second vector g_2 as in g_1.
(3) Apply Yanasse and Soma algorithm for the remainder $n - 2\log_2 c$ objects, but using g_1 already mounted in (1) as initial values.
(4) As in the *two-list* algorithm, search g_1 in ascending order and g_2 in descending order, adding your elements to find the optimal solution.

Next, it is presented a parallelization of this algorithm in a CREW PRAM. First, it is considered that the number of processors is $p = 2^q$, where $\log_2 \log_2(n - 2\log_2 c) \leq q \leq \log_2(n - 2\log_2 c)$. This interval on the quantity of processors is due to the use of *Algorithm* 2a in the parallelization proposed.

Similar to the Soma and Toth algorithm, the *Algorithm* 3 has four phases:

Phase (1): **Generation of** g_1 **by combining** $\log_2 c$ **weights.** Without any loss of generality, let these $\log_2 c$ weights be $w_0, w_1, \ldots, w_{\log_2 c-1}$. This phase has four steps as given next. Notice that initially a list L of size c is generated, which will store all the combinations of these $\log_2 c$ weights in a non-decreasing order.

Step 1: Generate a list L, void in the beginning, of all the combinations of weights $w_0, w_1, \ldots, w_{q-1}$. To do so, each processor P_i, $0 \le i < 2^q$, generates the sum of the weights w's corresponding to its index i expressed in binary. Moreover, to every generated sum, an index *ind* of the largest weight present in this sum is stored.

Step 2: Sort list L in increasing order by using 2^q processors. If two sums have the same value, consider the least value of *ind*.

Step 3:

```
for $i \leftarrow q$ to $\log_2 c - 1$ do in parallel
    (3.1) Copy list $L$ in another list $L'$ and add $w_i$
          to every element of $L'$, evaluating the
          new indices in ind.
    (3.2) Perform a parallel merge among lists $L$
          and $L'$, storing the resulting list in $L$.
end for
```

To the parallel merge, the value of *ind* is used to break ties whenever there are two equal sums.

Step 4: Let $L = [l_0, l_1, \ldots, l_{c-1}]$ be the generated list in the previous step, where $l_k = (l_k^1, l_k^2)$, $0 \le k < c$. l_k^1 is the value of the sum of the weights and l_k^2 is its corresponding value of *ind*, $0 \le k < c$. Each processor P_i, $0 \le i < p$, will write l_j^2 in $g_1[l_j^1]$ provided that $l_j^1 \ne l_{j-1}^1$, where $i\lceil c/p\rceil \le j < \min\{(i+1)\lceil c/p\rceil, c\}$. To avoid writing conflicts in vector g_1, P_i tests at the beginning if $l_{i\lceil c/p\rceil}^1 \ne l_{i\lceil c/p\rceil-1}^1$.

Clearly, *Step* 1 can be performed in time $O(q)$. For *Step* 2, it is possible to use an $O(q)$ algorithm [5]. However, as it is shown next, other parallel sorting algorithms of time $O(q^2)$ [2] could also be used, because this will not affect the time complexity of this phase.

To evaluate the complexity of *Step* 3, the commands interactions in **(3.1)** and **(3.2)** are analyzed separately. The first iteration in **(3.1)** is executed in constant time since there are $p = 2^q$ processors and a list with 2^q elements: every processor P_j, $0 \le j < p$, should add the value of w_i to the j-th element of list L'. In the following step, this time will double and so on, up to the last step. Therefore, the overall time of command **(3.1)** is $\sum_{k=0}^{\log_2 c-1-q} O(2^k) = O(2^{\log c - q}) = O(c/p)$.

Additionally, it is well known that, to a CREW PRAM with p processors, two sorted lists of size k can be merged in a time $O(k/p + \log k)$ [1]. Moreover, it is possible to observe that command **(3.2)** is executed $\log_2 c - q + 1$ times and, since there are $p = 2^q$ processors, it is immediate to conclude that the total time spent with these iterations in *Step* 3 is $O(2^{\log_2 c - q} + \log^2 c - q^2) = O(\frac{c}{p} + \log^2 c)$. Finally, since *Step* 4 is bounded by $O(c/p)$, **Phase (1)** can be evaluated in a time $O(\frac{c}{p} + \log^2 c)$.

Phase (2): **Insertion of** $n - 2\log_2 c$ **weights in** g_1**.** Once vector g_1 contains all the feasible combinations of the first $\log_2 c$ weights, the insertion of additional $n - 2\log_2 c$ weights can be done in parallel with $\log_2(n - 2\log_2 c) \le p \le n - 2\log_2 c$ processors through *Algorithm* 2a in time $O(\frac{(n-2\log_2 c)(c-2w_{\min})}{p} + \frac{c}{p} + n - \log c^2)$.

Phase (3): **Generation of** g_2 **with the combinations of the last** $\log_2 c$ **weights.** This phase is similar to **Phase (1)**. However, differently from vector g_1, g_2 will be the sorted list with c elements, *i.e.,* the list L of **Phase (1)**. As presented before, this list g_2 can be obtained in a time $O(c/p)$.

Phase (4): **Search of the optimal solution.** It is used an analogous procedure of Sanches et al. [16]. Initially, a *Prune Phase* is executed: it consists in splitting both lists in p equal size blocks and selecting pairs of blocks (one for each list) which can contain the optimal solution. With $p = 2^q$ processors, the *Prune Phase* spends a time $O(\log p) = O(\log n)$ and it selects at most $2p$ pairs of blocks. The sole difference with the Sanches et al. [16] algorithm is that, instead of two sorted lists, there is a vector g_1 and a sorted list g_2. Moreover, it is worth of mentioning that g_1 and g_2 store data of distinct nature: in g_1 there are indices of weights (values ranging from 0 to n, this latter one if no combination of weights is found for that knapsack capacity); on the other hand, in g_2 it is stored sums of weights.

Still, both structures possess the same size c and, for the *Prune Phase*, it is necessary that every processor P_i, $0 \le i < p$, detects first the largest and the smallest element of the i-th block of vector g_1. This can be achieved in a time proportional to $O(c/p)$, that is the size of the blocks. From that point on, simultaneous binary searches can be performed in the list g_2, throughout the remaining of the *Prune Phase*.

Finally, every processor has associated to itself one or two pair of blocks. Since these blocks have size proportional to $\lceil c/p\rceil$, this phase can be executed in a time $O(c/p)$.

Therefore, the total time of **Phase (4)** is $O(\frac{c}{p} + \log n)$.

Overall Complexity. The four phases presented before perform a total time of $O(\frac{c}{p} + \log^2 c + \frac{(n-2\log_2 c)(c-2w_{\min})}{p} + n - \log c^2 + \log n) = O(\frac{(n-2\log_2 c)(c-2w_{\min})}{p} + \frac{c}{p} + n - \log c^2 + \log^2 c) = O(\frac{n}{p}(c - 2w_{\min}))$, and this is the final time complexity of *Algorithm* 3. In terms of space, just $O(n + c)$ is necessary.

Table 1
Parallel algorithms

Algorithms	PRAM	Time	Processors
Algorithm 1	CREW	$O(\frac{n}{p}(c - w_{\min}) + \frac{c+w_{\max}-w_{\min}}{p} + n + \log p)$	$p \leq w_{\min}$
Algorithm 2a	EREW	$O(\frac{n}{p}(c - 2w_{\min}) + \frac{c}{p} + n)$	$\log_2 n \leq p \leq n$
Algorithm 3	CREW	$O(\frac{(n-2\log_2 c)(c-2w_{\min})}{p} + \frac{c}{p} + n - \log c^2 + \log^2 c)$	$\log_2(n - 2\log_2 c) \leq p \leq n - 2\log_2 c$

5. Conclusions

The first scalable parallel algorithms for solving a variation of the Knapsack Problem – SSP with n objects and capacity c – in time $O(\frac{n}{p}(c - w_{\min}))$ and space $O(n + c)$ were presented. They were designed to a PRAM with p processors and improve the current time and space *upper-bounds*.

Table 1 summarizes the computational complexity time of these algorithms, with the respective PRAM model and the number of processors being used.

Acknowledgements

The first author's work was partially financed by FAPESP (grants 99/09483-5 and 06/05325-1).

References

[1] S.G. Akl, The Design and Analysis of Parallel Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1989.
[2] K.E. Batcher, Sorting networks and their applications, in: Proceedings of AFIS 1968 SJCC, 32, Montvale, NJ, 1968, pp. 307–314.
[3] R.E. Bellman, Dynamic Programming, Princeton University Press, Princeton, 1957.
[4] Chung,Vijaya, A randomized linear-work EREW PRAM algorithm to find a minimum spanning forest, Algorithmica 35 (2003) 257–268.
[5] R. Cole, Parallel merge sort, SIAM J. Comput. 17 (1988) 770–785.
[6] A.K. Datta, R.K. Sen, $o(\log n)$ time parallel maximal matching algorithm using linear number of processors, Parallel Algorithms and Applications 19 (2004) 19–32.
[7] A. Goldman, D. Trystram, An efficient parallel algorithm for solving the knapsack problem on hypercubes, Journal of Parallel and Distributed Computing 64 (2004) 1213–1222.
[8] P.S. Gopalakrishnam, I.V. Ramakrishnam, L.N. Kanal, Parallel approximate algorithms for the 0–1 knapsack problem, in: Proceedings of International Conference on Parallel Processing, 1986, pp. 444–451.
[9] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, Journal of ACM (1974) 277–292.
[10] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, 2004.
[11] G.A.P. Kindervater, J.K. Lenstra, An introduction to parallelism in combinatorial optimization, Discrete Applied Mathematics 14 (1986) 135–156.
[12] J. Lee, E. Shragowitz, S. Sahni, A hypercube algorithm for the 0/1 knapsack problem, Journal of Parallel and Distributed Computing 5 (1988) 438–456.
[13] J. Lin, J. Storer, Processor efficient hypercube algorithm for the knapsack problem, Journal of Parallel and Distributed Computing 3 (1991) 332–337.
[14] E.W. Mayr, Parallel approximation algorithms, in: Proceedings of International Conference on Fifth Generation Computer Systems, 1988, pp. 542–551.
[15] S. Rajasekaran, Efficient parallel hierarchical clustering algorithms, IEEE Transactions on Parallel and Distributed Systems 16 (2005) 497–502.
[16] C.A.A. Sanches, N.Y. Soma, H.H. Yanasse, An optimal and scalable parallelization of the two-list algorithm for the subset-sum problem, European Journal of Operational Research 176 (2007) 870–879.
[17] N.Y. Soma, P. Toth, An exact algorithm for the subset sum problem, European Journal of Operational Research 136 (2002) 57–66.
[18] S. Teng, Adaptive parallel algorithms for integral knapsack problems, Journal of Parallel and Distributed Computing 8 (1990) 400–406.
[19] Y.R. Wang, S.J. Horng, An O(1) time algorithm for the 3D Euclidean distance transform on the CRCW PRAM model, IEEE Transactions on Parallel and Distributed Systems 14 (2003) 973–982.