0-1 Knapsack Problem in parallel

Progetto del corso di Calcolo Parallelo
AA 2008-09

Salvatore Orlando

0-1 Knapsack problem

- N **objects,** $j=1,..,N$
- **Each kind of item** j **has a value** p_j **and a weight** w_j **(single dimension)**
- **You can fill a knapsack, with an integer weight capacity of** W
- **How much worth (sum of values) can you transport in one trip?**

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq W, \qquad x_j \in \{0,1\}, \quad j = 1, \ldots, n.$$

Special case decision problem:

- **weights equals to values:** $w_j = p_j$
- ***Given a set of nonnegative integers, does any subset of it add up to exactly W?***
- **Equivalent to the** ***subset sum problem***

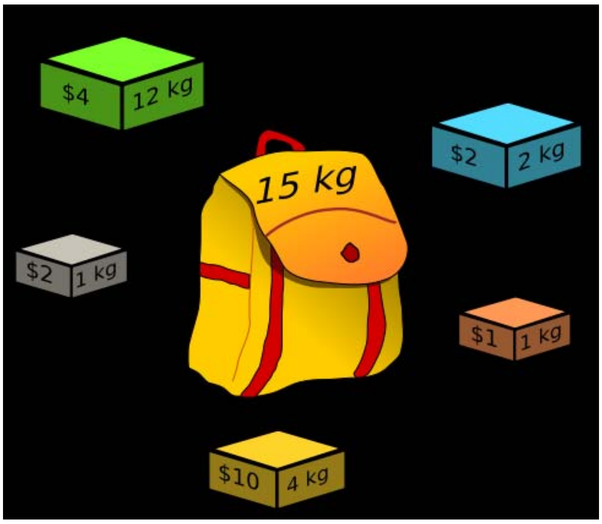
Bounded Knapsack problem

- **There is a a maximum integer value b_j of item j available to fill the knapsack.**

$$\text{maximize} \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to} \sum_{j=1}^{n} w_j x_j \leq W, \qquad x_j \in \{0, 1, \ldots, b_j\}, \quad j = 1, \ldots, n$$

Knapsack



Single dimension problem: only weights

Which boxes should be chosen to maximize the amount of money, while still keeping the overall weight under or equal to 15 kg?

Multi dimensional problem: also consider the density or dimensions or …. of the boxes.

Solution bounded knapsack problem : 3 yellow boxes and 3 grey boxes
Solution 0/1 knapsack problem : all boxes but the green one

0-1 Knapsack

- The obvious naïve solution consists in trying every possible combination
  - 2^N combinations

- A slightly better method is branch-and-bound
  - breadth-first search of the combination space, but prune branches that cannot lead to optimal solutions.

- A better solution follows from expressing the problem as a recurrence relation and taking a dynamic programming approach
  - *Dynamic programming* algorithmic technique is based on the knowledge of *optimal solutions* for *subproblems*
  - This knowledge is used to find the optimal solutions of the overall problem algorithm
  - *Dynamic programming* algorithms store information about common subproblems in a table
  - Fill the table until you reach the solution.

0-1 Knapsack: Dynamic Programming

- Weights are positive

- Dynamic programming table
  - let $A(j, Y)$ be the maximum profit that can be attained for the *subproblem* with weight less than or equal to Y using items from 1 to j.
  - then $A(N, W)$ if the maximum profit of the overall problem

- Solved in *pseudo-polynomial time*
  - its running time is polynomial in the *numeric value* of the input (which is exponential in the *length of the input* -- its number of digits).
  - in this case $O(N W)$, where the size of input W is $\log W$
  - $O(N\, 2^{\log W})$

0-1 Knapsack: Dynamic Programming

- $N \times W$ table A, indexed by an item number and a knapsack capacity
- We can define $A(j,Y)$ recursively as follows:

$$A(0, Y) = 0$$

$$A(j, 0) = 0$$

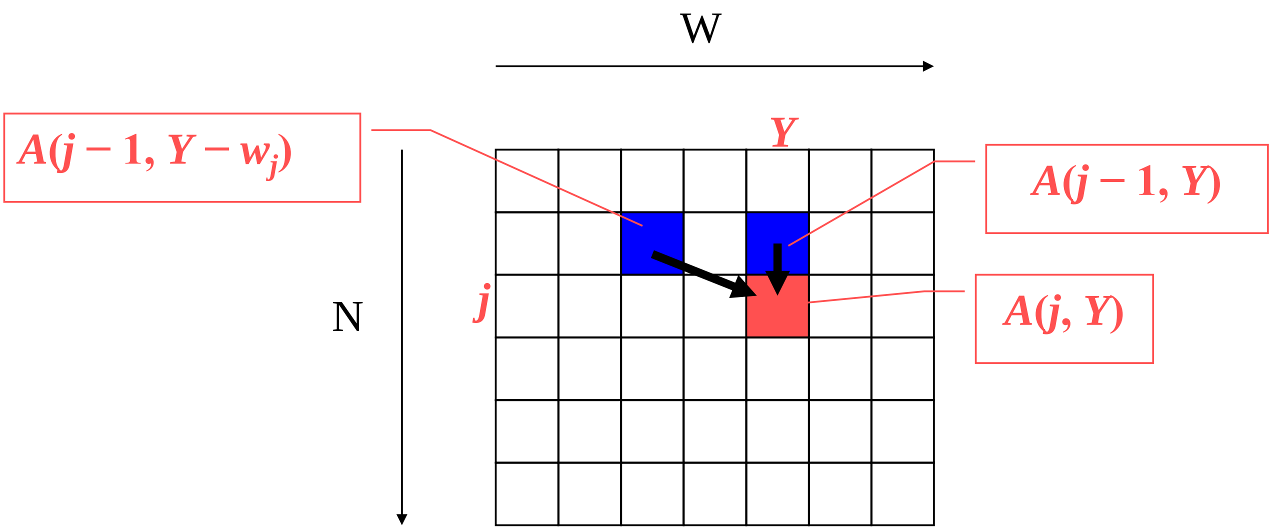
Item j cannot be inserted in the knapsack

$$A(j, Y) = A(j-1, Y) \quad \textbf{if } w_j > Y$$

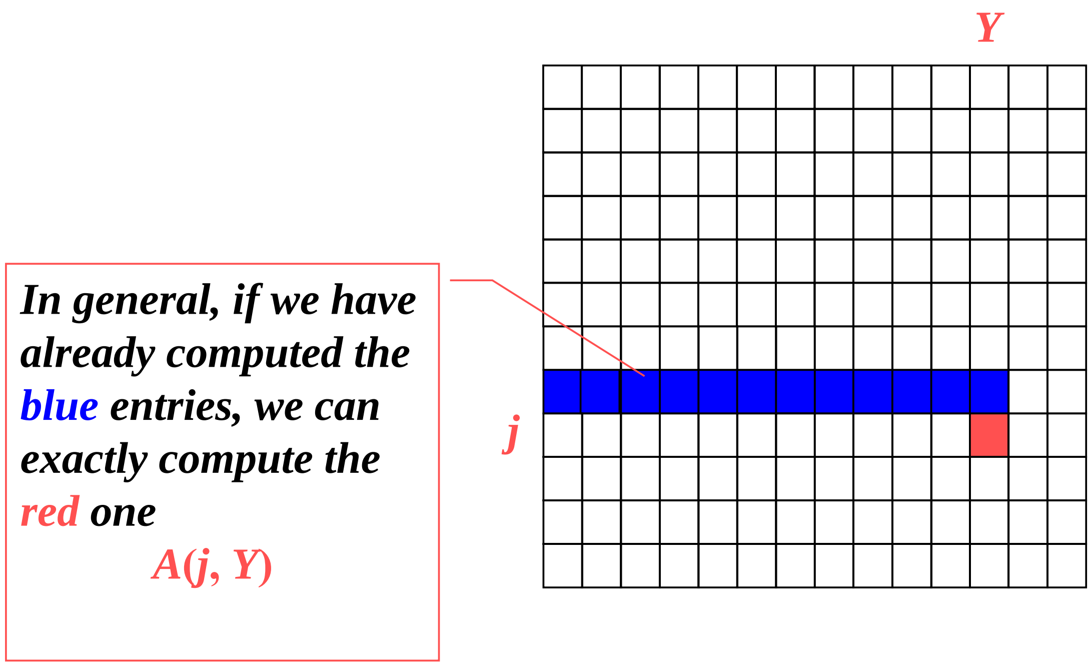
$$A(j, Y) = \max \{ A(j-1, Y), \quad p_j + A(j-1, Y - w_j) \} \quad \textbf{if } w_j \le Y$$

Either item j is not considered, or j is inserted in a knapsack (of weight capacity of $Y - w_j$) optimally filled using items 1 through j-1

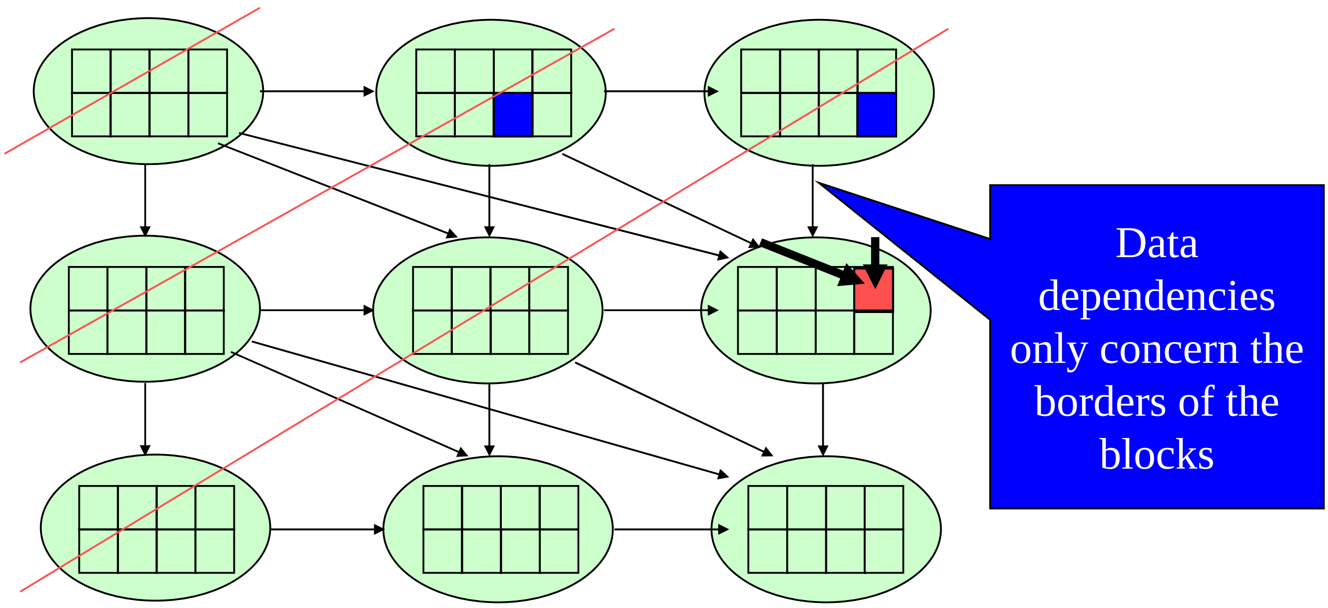
Dependencies



Dependencies

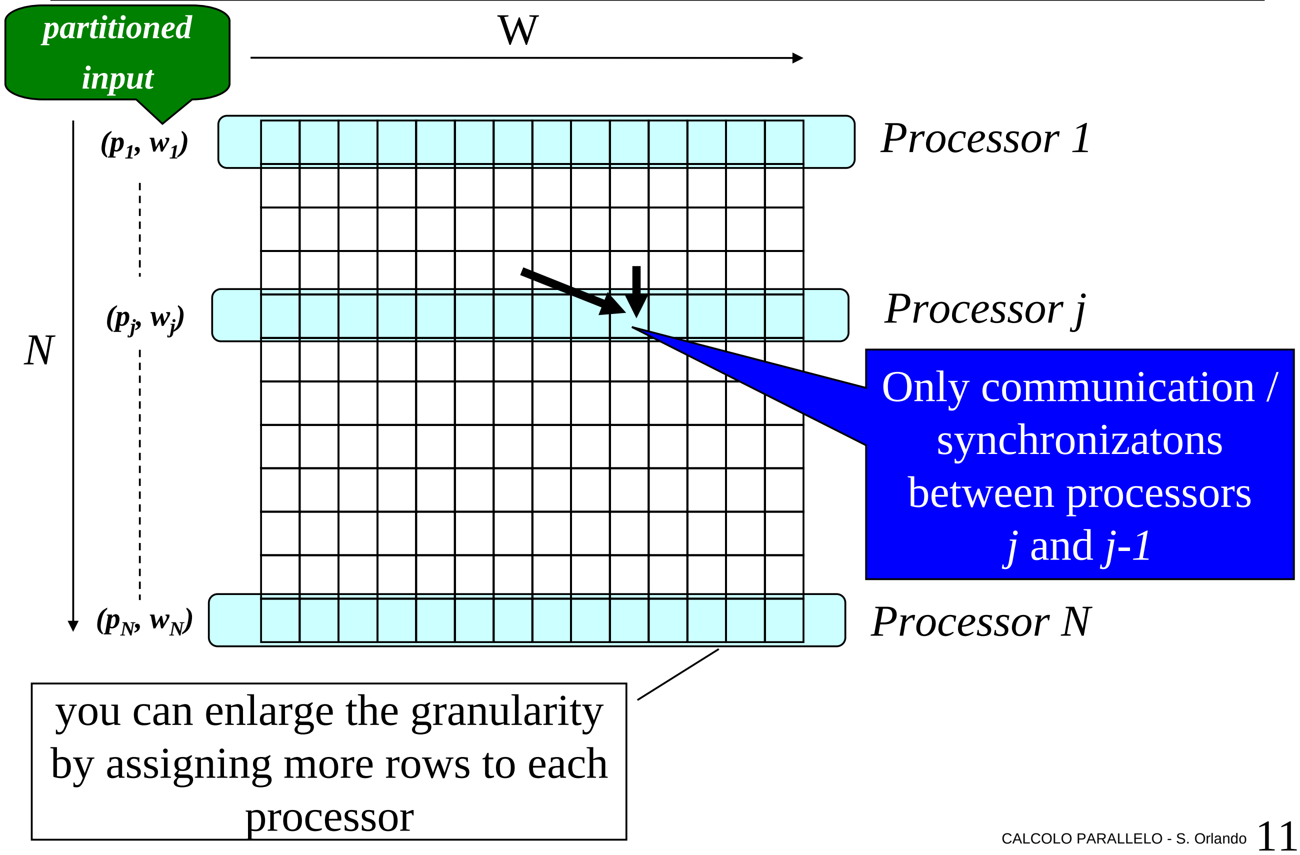


Example of Task Dependency Graph (obtained by partitioning the output)

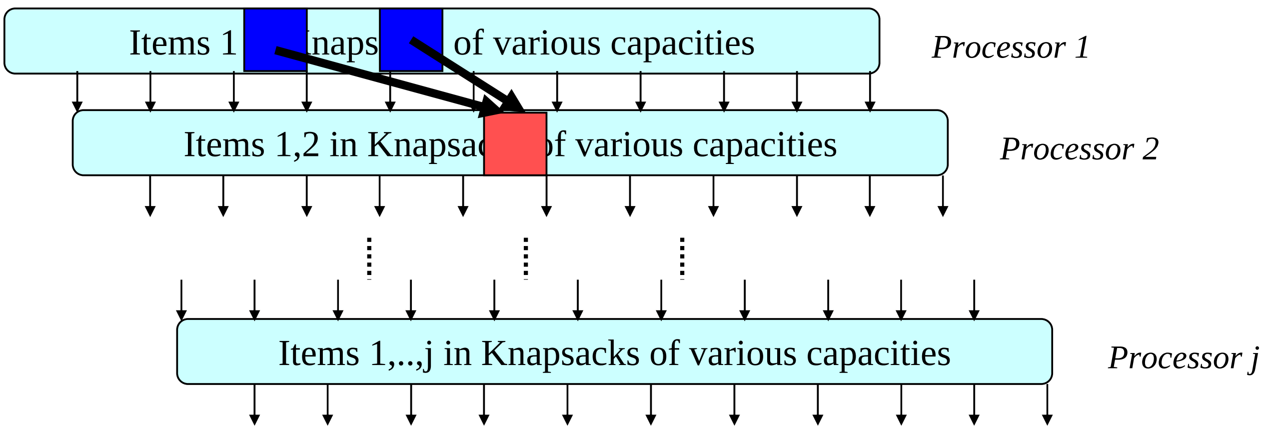


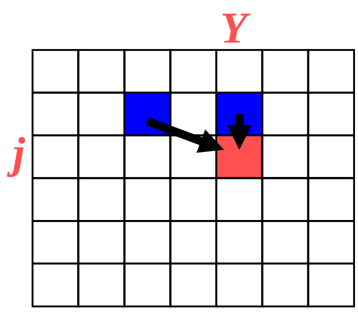
- We fixed a average granularity of task (unit of scheduling)
- Note the data dependencies
- Can be implemented in either shared memory or message passing

Example of mapping (partitioning the output)

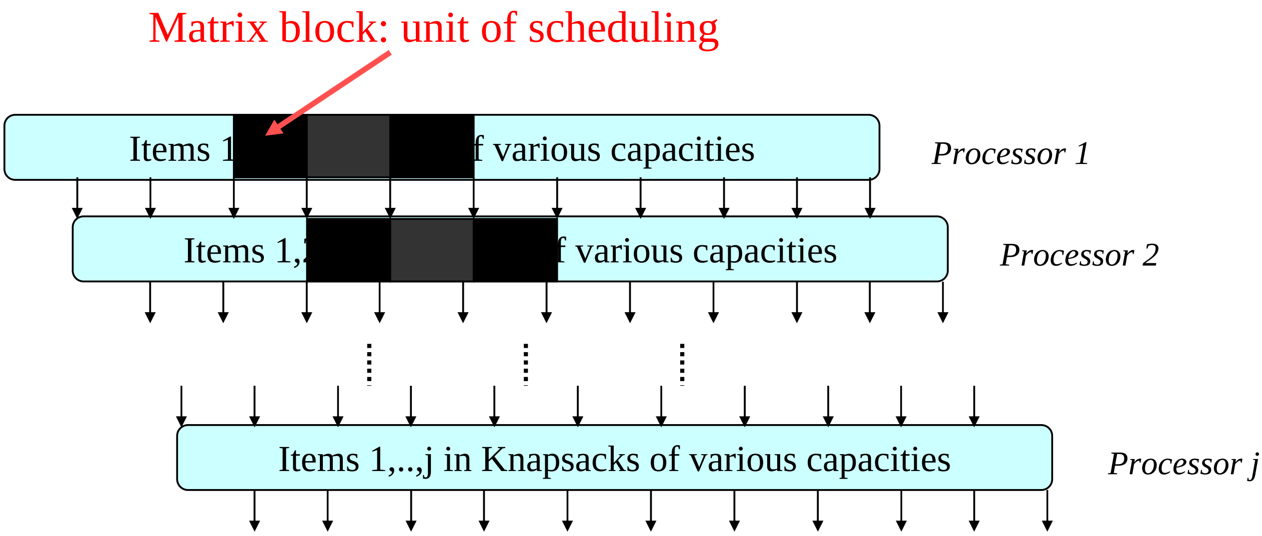


Pipelining

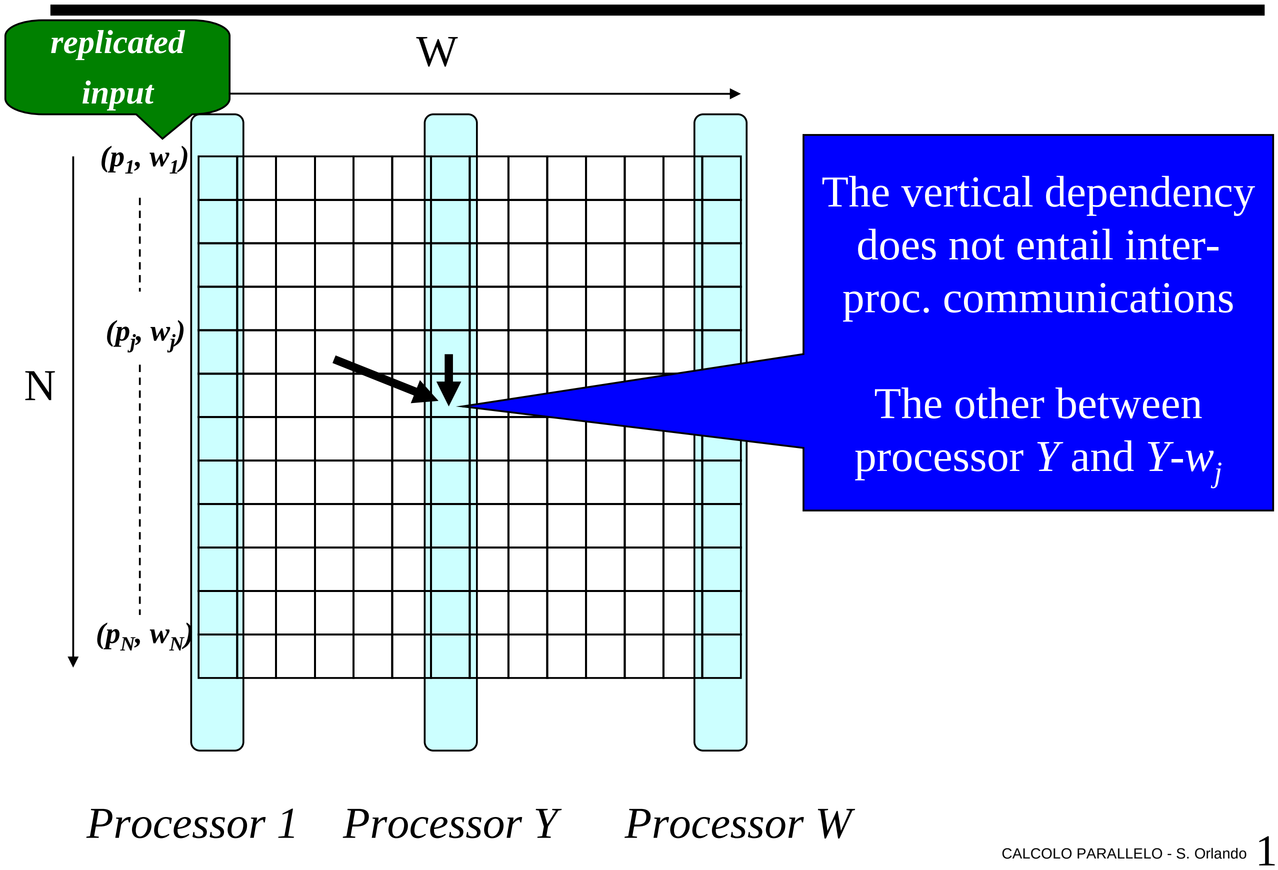




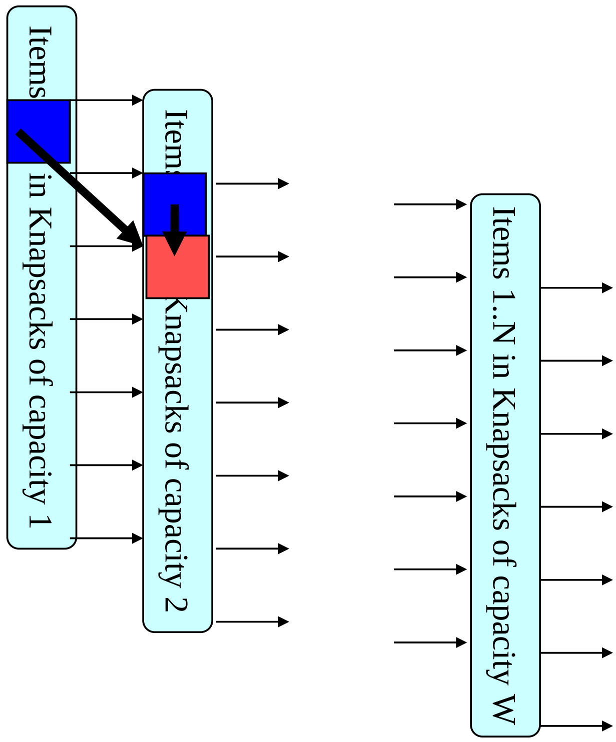
Pipelining



Example of mapping (partitioning the output)



Pipelining



What is requested

- **Write a parallel 0/1 knapsack solver, using POSIX threads and/or MPI**
- **Unit of scheduling can be assigned *statically* or *dynamically* (in the last case, also ensuring *load balancing*)**
- **The program must produce:**
  - **The optimal profit**
  - **A *binary vector Objs*, representing the objects chosen (1) and not chosen(0)**
  - **Total weight of the chosen objects**
  - **Running time in seconds.**

    Example output for 3 objects:
    ```
    Profit: 12
    Objs: 011
    Weight: 7
    Time: 0.011 seconds
    ```
- **It is important to evaluate**
  - **speedup**
  - **scalability, i.e. how does the overall completion time change by increasing both the problem size and the number of processors?**

What is requested

- A short but complete description of the parallel solutions and tradeoffs
- A short but complete discussion on the performance of your programs
  - Comparison of sequential and parallel run-times
  - Speedup and scalability issues
  - Evaluating the changes of the input data, task granularities, number of processors employed, etc.

- Write a small scientifis report … in English !?
  - abstract, small introduction, problem statement, projects of the various parallel solutions, performance evaluation, conclusion that summarize main results)
- Prepare a short presentation supported by slides to present the project

Generator of Knapsack problems

- http://www.diku.dk/~pisinger/generator.c

- generator n r type i S

- n: number of items
- r: range of coefficients p_j and w_j
- type
  - 1=uncorr. (p_j and w_j randomly distributed in [1, R])
  - 2=weakly corr. (w_j randomly distributed in [1, R], and $p_j \geq 1$ randomly disributed in [w_j – *R/10, w_j* + *R/10*])
  - 3=strongly corr. (w_j randomly distributed in [1, R], and $p_j = w_j$ +*10*)
  - 4=subset sum (w_j randomly distributed in [1, R], and $p_j = w_j$)
- i: instance no
- S: number of tests in series (typically 1000)

- i and S determine the capacity W of theproblem instance

Knapsack problem instances

```
orlando@ihoh:~/knapsack$ ./a.out
generator
n = 10
r = 4
t = 1
i = 1
S = 1000

orlando@ihoh:~/knapsack$ less test.in
10                    // N number of objects
    1     2     1     // 1     p1     w1
    2     2     2     // 2     p2     w2
    3     4     4     // 3
    4     4     1
    5     1     2
    6     1     3
    7     1     3
    8     3     1
    9     2     3
   10     2     4
5                     // Knapsack capacity W
```

How to create the problem instances for various *n*

- **r = n/10**
- **t = 1**
- **i = 7**
- **S = 1000**