Design and implementation of a parallel algorithm

0/1 Knapsack problem

ABSTRACT

The course aims to give students a throughout knowledge about parallel and distributed systems. It covers general issues of parallel and distributed processing from a user's point of view which includes system architectures, programming, performance evaluation, applications, and the influence of communication and parallelism on algorithm design. A final project concludes the course by making students work about their choice (turned on implementation or research) and put into practice the knowledge gained during the course. Our project consists on designing and implementing a parallel algorithm to solve the knapsack problem. Based on the sequential algorithm, 3 programs have been implemented using MPI, openMP and Hadoop. Finally, test performance has been performed to compare the solutions.

Keywords: Hadoop, OpenMP, MPI, Map Reduce

INTRODUCTION

This project aims to put into practice the knowledge gained during the semester in the Parallel and Distributed Processing course. We choose to work on the 0/1 Knapsack problem, and to implement two solutions using parallelism to solve this problem.

This subject has been chosen since we wanted to work on a programming project to gain skills and practice on parallel implementation methods learnt during the semester. The 0/1 knapsack problem seemed to be a good subject as it was a problem on which neither of us had worked before, so we both were discovering it, and seemed to be understandable and interesting. Finally, we choose two work as a 2-student's team to be able to produce 3 programs and compare their efficiency.

Our work has been divided into several parts. The first one has consisted on analyzing and understanding the 0/1 knapsack problem and find a sequential algorithm. Then, we have worked simultaneously, to design and implement a MPI based solution, openMP based solution and a Hadoop based solution. Finally, the last part has been to create a benchmark to compare and analysis the efficiency of both programs.

This report relates this work and completes the presentation made in class.

RELATED WORK

A significant amount of research has been done on the 0- 1 knapsack problem, which has numerous applications in business. First scientists and mathematicians, like Balas or Zemel developed a sequential algorithm. More recently, parallel algorithms have been discussed by a number of people. Several approaches has been studied by mathematicians like large data set problem or CPU based solution. Martello pointed out the impracticality of an FPTAS for the knapsack problem due to this increase in space requirement

Our objective is not to challenge the results of previous researches but to familiarize ourselves with the openMP, MPI and Hadoop design and implementation, and to observe and analyze the consistency of our results.

The sequential algorithm for the problem has been reuse from prior work. Our part of the work has been to implement the sequential program, and to design and implement the 3 solutions we propose.

PROBLEM STATEMENT

1. Knapsack Problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization.

A knapsack can carry a fixed weight. Given a set of items, each with a weight and a value, the objective is to determine the number of item to include in the knapsack so that the total weight is less than or equal to the limit of the knapsack and the total value is as large as possible.

More precisely, we worked on the 0/1 version of the knapsack problem, in which an item is either put is the knapsack or not and, in contrast to the bounded and unbounded variants, cannot be multiplied or divided.

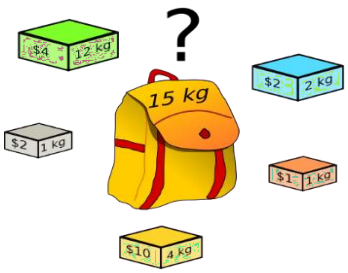


Figure 1: Bag Problem *Source: Wikipedia*

2. Sequential algorithm

To understand the sequential algorithm, we need to find a representation of the computations' evolution. To do so, we use a 2-dimension matrix as follow:

	1	2	3	...	Max weight
Item 0					
Item 1					
...					
Item n-1					

Thus, at the end of the computations, the best value achievable for the input data is found is the cell [item n-1][max weight].

Then, the sequential algorithm is:

Figure 3: Pseudo code for sequential knapsack problem
Source: Wikipedia

We use a 2 for-loops structure to parse the matrix with a row orientation (the loop handling rows is the outer loop).As described above, each cell is filled with the maximum achievable value for its column weight and row (and below) items. So at each iteration, the value is the maximum value between:

- The value computed for the same weight without using the new item. This value is found in [i-1][j] (same column, upper row)

Figure 2: Problem representation

Each cell contains the best value achievable:

- using the item of its row and the items above it
- not exceeding the weight of its column

- The value using this item and the best value achievable for the weight: ‘column’s weight minus item’s weight’. This value found in [i-1][j- item’s weight].

That is the computation made in the line 15. The computation line 13 is executed if the weight of the considered item is higher than the column weight. In that case, the best value achievable is the value of the row above. The first loop is used to fill the first row with 0 as a reference for the next computations.

At the end of the computations, the best value is found in the cell [item n-1] [max weight]. To know witch items is used, a back sparse should be done.

METHODOLOGY

1. Dependencies analysis

First, our project consists on software parallelism so we didn't consider hardware parallelism.

While analyzing the above algorithm, we can first conclude that there is no task parallelization realizable. Indeed, all the task depend of each other and we cannot extract some computation that can be done simultaneously.

We can first conclude that we will parallelize our solution using data parallelism. Since, the same computation is done on every cell of the data, this seems to be the more obvious solution.

Then, we need to figure out which data is parallelizable. As we analyze the sequential algorithm, we saw that the value of a cell can either be the value of the cell [i-1] [j] or use the data in the cell [i-1] [j- item's weight]. We can then conclude that there are dependencies between the rows but not on the columns. Indeed, a cell need to know the value of the rows before itself, but do not need the value of the column of its rows.

So, the code is organizing with a column parallelization. That means that for each row, all the column is performed simultaneously.

2. MPI optimization

The MPI implementation use the column parallelization explained above. Each column of the matrix is mapped to a processor. The mapping is made to evenly divide the work, so each process works on the columns (process number + k*number of process).

For each row, the rank computes the column it is mapped to. At each iteration the execution contains several steps:

1. Compute the maximum value achievable using the item of the row
    a. If the weight of the item is bigger than the column's weight, the value is 0
    b. If it's the first item and the If the weight of the item is smaller than the column's weight, the value is the item's value
    c. Else, the value id the item value plus the value of the cell [i-1][j-weight[i]]. The rank gets this value thanks to a MPI_Recv from the rank that had compute this value.
2. Compute the value without the new item. This value is the value just above in the matrix (same maximum weight withut the new item) or 0 if it is the first item.
3. Save in the cell the maximum value achievable using or not the new item
4. Send to all the processers that could need it in future iteration the new value.

This implementation is quite optimized since, it does require any "waiting time". Indeed, there is no need for MPI_Barrier, sending are non-synchronous and receiving is fast since the data must have been send before.

3. OpenMP Optimization

OpenMP optimization uses the column parallelization. Each column of the matrix is mapped to a processor. The mapping is made to evenly divide the work, so each process works on the columns (process number + k*number of process).

There are several steps in execution of OpenMP version.

- First, a new instance is generated for the knapsack problem. System ask for the user to enter the four parameters `N`: Total number of items. `max_`: The maximum weight and value of an item. `w`: Array of weights. v`: Array of values.
- As shown above in the code here we tried to parallelize the calculation of the every column. So once all the weights are calculated column wise then only thing left is to compare all the weights and find out the best profit.
- The main improvement over the sequential version is that it only uses an array of two rows because for every element to be computed, we only need two elements from the previous row.
- Second method is use to get the total weight and calculate the total profit.

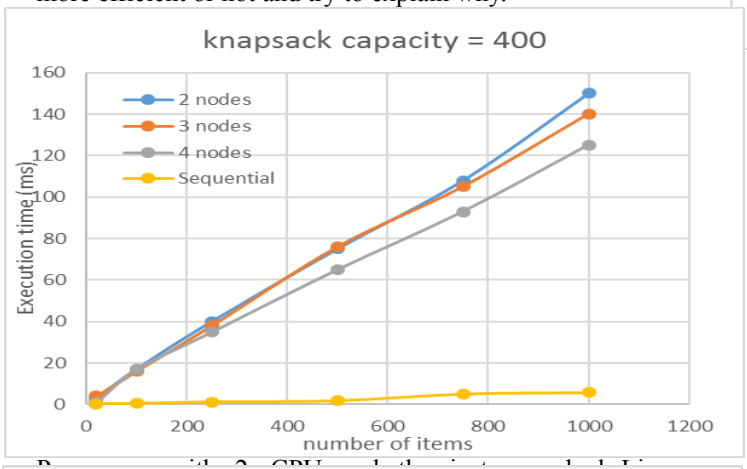
4. Hadoop Optimization

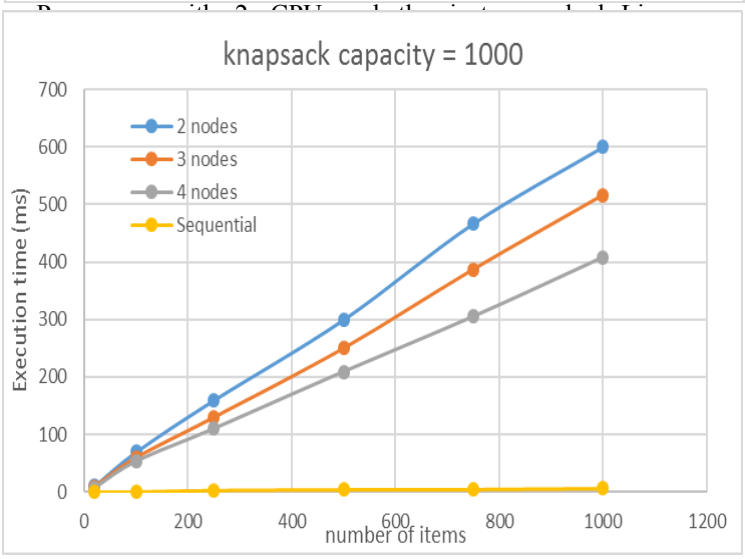
The first step has been to set up the cluster. Please refer to the annexes for the described steps of the cluster set up.

Unfortunaly, we didn't achieve to run our code on the cluster. We thus are not able to have result from our Hadoop implementation.

EXPERIMENTATIONS

After implementing the programs, the second part of the project was to stress the program in order to see if it was more efficient or not and try to explain why.





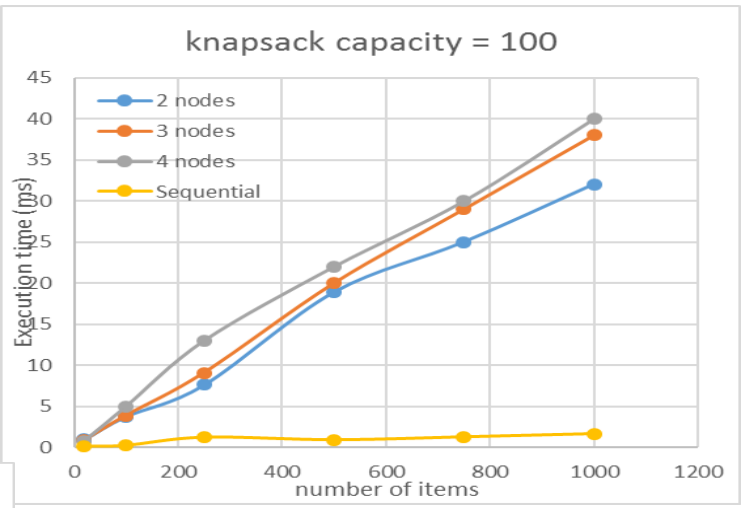
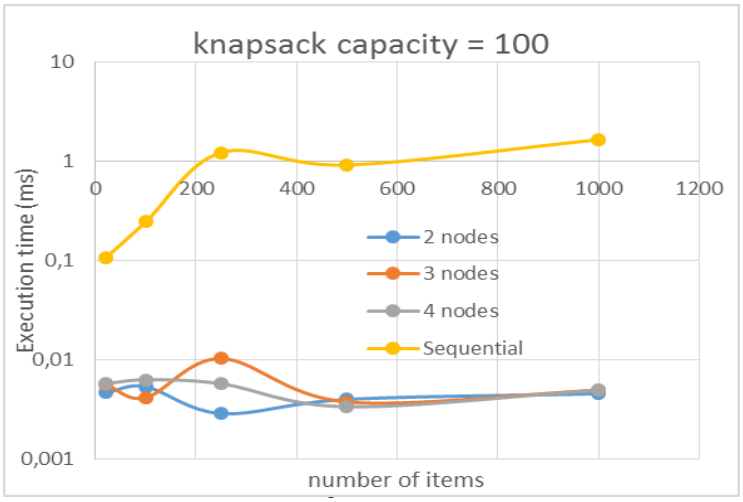


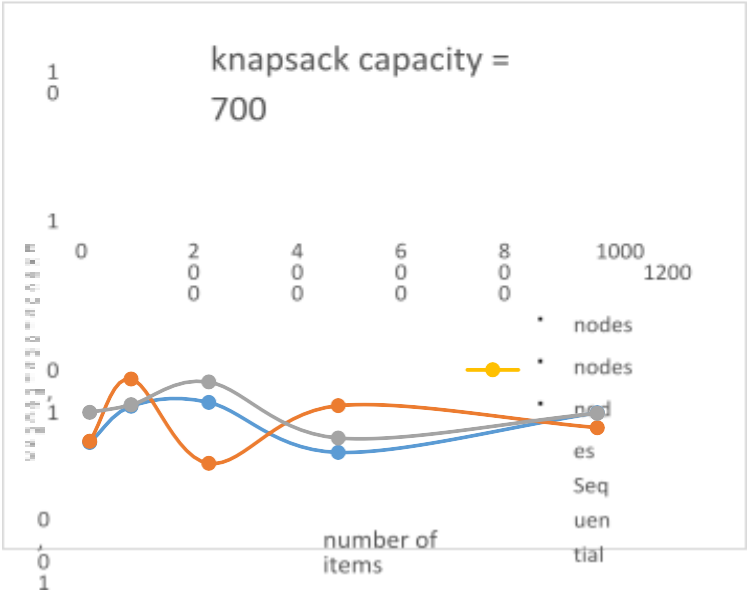
Figure-3: MPI execution results



test ou program on bigger data set since our computers was not powerful enough. It is possible that sequential execution time increases exponentially since MPI execution time regulates.

Nevertheless, we can still analyze that for smaller matrix (when the knapsack capacity is under 200), the execution time is bigger with 4 nodes than with 2 nodes. This can be explain since "starting" the program by initializing MPI take times. And this time don't make it profitable if the amount of data to compute is too small.

Given the results, we didn't compute the speed up.



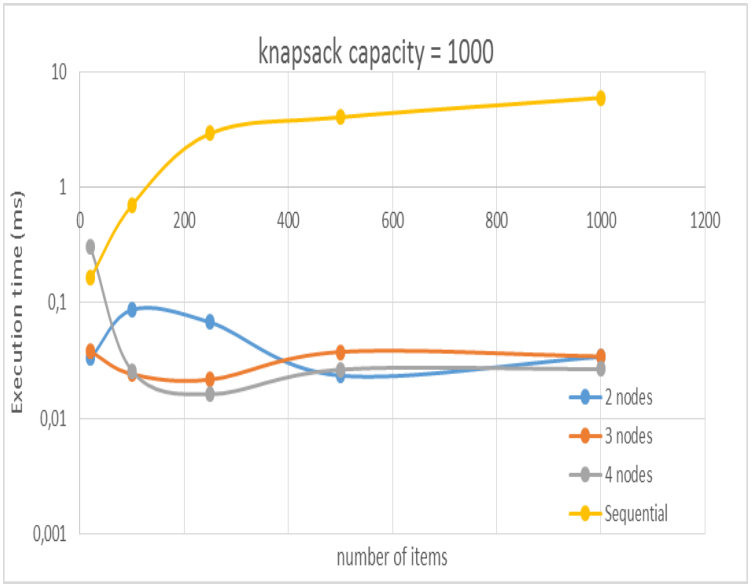


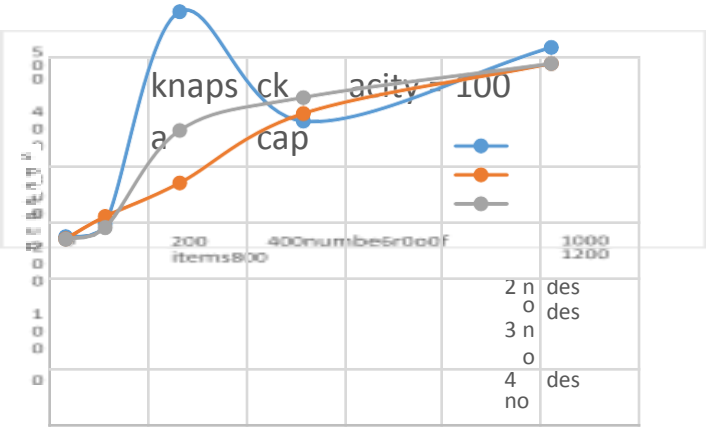
Figure-4: openMPI execution results

So, we can first see that our OpenMP program is fully optimize the execution time at all. Even on big matrix like 1000*1000, OpenMP algorithm is hundreds of times better than sequential. We can think to several points that could explain this:

Column indices can be solved independently in the knapsack 0/1 version. So applying pragma for directory to the column loop. So that each column execution is done parallelly.

We can also notice that some measures doesn't seem coherent. These measure are not to take into account on our analysis, since we noticed irregularities during our tests, and some measures may not be representative.

Then, we computed the speedup:



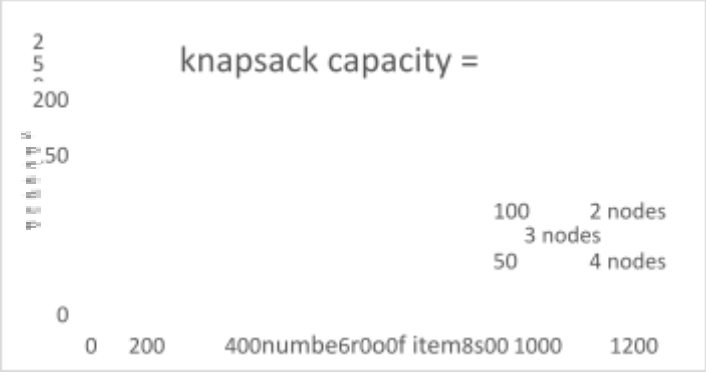


Figure5: Speed up for openMP

So we observe a very good speed up (between 100 and 400) that was expected regarding the results. Moreover, we observe that the speed up is not constant, thus, it gets better when the matrix size increases. We can think that it is because, compute a small amount of data with a sequential program is efficient but the efficiently decrease exponentially with the matrix expansion. As it takes some time to initialize the openMP library, more the matrix increases, more this initialization time is make profitable.

c) Influence of knapsack capacitiy and number of items

Finally, we "cross the data" to try to determine what have the bigger influence on the execution time: the knapsack capacity or the number of item.

To test that we compared the execution time for matrix of the same rough size, but with high number of items and small knapsack capacity and vice versa.

The results are:

Figure6: Influence of capacity and number of items

So we can observe:

- For the sequential program: compute a high number of items takes more time
- For the openMP and MPI program: compute a high capacity takes more time

This observations are consistent with the way we paralyzed our system. Indeed, the columns (which are the capacity values) are parallelized and computed simultaneously. So a matrix with a small number of column will be compute fast since each thread will have few work at each iteration. On the other side, as the capacity increases, the amount of work per thread at each iterations also increases and the total execution time become longer

FUTURE ENHANCEMENTS

There are several way this work could be extended in the future.

- The first objective would be to achieve to run the Hadoop program.
- Then, the MPI program should be optimized, and think over to determine where delays appears and how we could reduce them.
- In order to stress ours program, bigger data set could be used to determine the efficiency of our programs on extra-large matrix.

- Our programs could also be run and tested on more powerful computers with an higher number of nodes.
- Finally, another approach could be to compare develop another solution to this problem and to compare the efficiency. For example, a CPU based solution, using CUDA.

.

CONCLUSION

To conclude this project was very interesting since it let

us reflect and work on a practical parallelism problem. We have put into practice the knowledge learnt during this course, both technically with the program implementations and theoretically with the analyses of results.

We managed to design and implement 3 programs using MPI, openMP and Haddop. Even if some results were not what was expected, it let us reflected about the reasons of this and some possible enhancements. OpenMP provided good results and thus we were able to conclude that the way we parallelized the problem was correct. Finally, analyzing the influence of capacity and number of items was very interesting since we were able to report consistent results for parallelized programs.

[4] 4. “An efficient parallel algorithm for solving the knapsack problem on the hypercube” A. Goldmany and D. Trystram

[5] 5. “Approximation algorithms for minimum knapsack problem” Mohammad Tauhidul islam

[6] 6. “Parallelization of the Knapsack Problem as an Introductory Experience in Parallel Computing” Michael Crawford, David Toth

[7] 7. “0-1 Knapsack Problem in paralle”l Progetto del corso di Calcolo Parallelo, Salvatore Orlando