Computational Sciences and Engineering 4(1) (2024) 31-40

University of Guilan

Computational Sciences and Engineering

journal homepage: https://cse.guilan.ac.ir/

GiraDP: Enabling Large-Scale Optimization with Dynamic Programming

Saeed Mirpour Marzuni [a], Javad Vahidi [b,*]

[a] Department of Electrical and Computer Engineering, University of Science and Technology of Mazandaran, Behshahr, Iran
[b] Department of Computer Science, Iran University of Science and Technology, Tehran, Iran

ARTICLE INFO

Article history:
Received 29 January 2025
Received in revised form 12 February2025
Accepted 12 February 2025
Available online 12 February 2025

Keywords:
Giraph
Dynamic
Programing
Hadoop
Knapsack Problem

ABSTRACT

NP-hard optimization problems are computationally challenging tasks that require significant resources to solve, particularly as problem sizes increase. In this paper, we introduce a novel framework, GiraDP, which converts dynamic programming problems into graphs and solves these large-scale graphs efficiently. GiraDP addresses NP-hard optimization problems by leveraging advanced graph processing techniques. The framework utilizes Giraph and Hadoop in its architecture to manage extensive datasets and complex computations. It generates input data for Giraph and identifies active vertices that correspond to sub-problems within the larger problem. Our experiments include scenarios with 20, 40, and 60 items, across varying knapsack capacities. The results indicate that while the serial algorithm performs best at low capacities, it fails to handle larger instances due to memory limitations, resulting in heap space errors. In contrast, GiraDP demonstrates superior efficiency and scalability for high capacities and large item sets. The MPI-based approach also shows improved performance over the serial algorithm for larger problems, although it does not match the efficiency of GiraDP. These findings underscore the importance of distributed computing solutions for large-scale optimization problems.

1. Introduction

One approach to solving optimization problems is dynamic programming. In many application domains, such as IoT and bioinformatics, it is an efficient solution. However, the time complexity of some dynamic programming problems, such as the knapsack problem and the traveling salesman problem, is exponential, making them NP-hard [1]. When the volume of data grows, the execution time increases, rendering these algorithms impractical for such problems. To address

[*] Corresponding author.
E-mail addresses: jvahidi@iust.ac.ir (J. Vahidi)

https://doi.org/10.22124/cse.2025.29701.1091
© 2024 Published by University of Guilan

big data in dynamic programming, some approximate algorithms have been introduced, but they do not produce optimal answers. Achieving optimal solutions in reasonable time frames necessitates parallelizing dynamic programming in a distributed manner to keep execution time at manageable levels.

There are two models for parallel programming: (1) the shared memory model [2] and (2) the distributed memory model [3]. The shared memory model is suitable for efficient data communication, while the distributed memory model offers better scalability at the cost of higher communication overhead. Graph processing systems provide a high level of scalability on fast, interconnected, high-performance computing machines. Their behavior on "virtualized commodity hardware," known as cloud computing, is also notable [4]. Frameworks such as Apache Giraph [5], GraphX [6], and GraphLab [7] offer powerful distributed algorithms for processing large graph data.

In this paper, we convert dynamic programming problems into graphs and then solve these problems using GiraDP, which processes large-scale graphs. Specifically, we introduce a new framework, GiraDP, that addresses NP-hard optimization problems through graph processing. GiraDP leverages Giraph and Hadoop in its architecture. It prepares the input data for Giraph and determines which vertices are active. Dynamic programming simplifies complex problems by breaking them down into simpler sub-problems in a recursive manner. This creates a relationship between the value of the larger problem and the values of the sub-problems. Using this concept, we solve sub-problems through the inner vertices of the graph, then transfer their results to the next step.

The rest of the paper is structured as follows: Section 2 briefly describes some background information and related work. In Section 3, we detail the GiraDP framework and its architecture. Our experimental results are reported in Section 4, and we provide conclusions and future research directions in the final section.

2. Background and related work

In this section, we begin by introducing Apache Giraph [5], the fundamental framework for our system. Then, we review studies conducted in the field of optimization problems in big data. The National Research Council of the National Academies of the United States [8] identifies graph processing as one of the seven computational giants in massive data analysis. Google’s Pregel [9] is the first graph processing framework in the literature to employ the bulk synchronous parallel (BSP) model [10] for graph computation using a vertex-centric approach. Public implementations of this framework include Giraph [5], GoldenOrb [11], Apache Hama [12], and others.

Apache Giraph is an open-source implementation of Google's proprietary Pregel framework, designed for distributed graph processing. It utilizes a cluster of machines (workers) to manage and process large-scale graph datasets efficiently. One of these machines functions as the master, coordinating tasks among the worker machines, known as slaves. The master's responsibilities include global synchronization, error handling, partition assignment, and the aggregation of values from different workers. Giraph is built on the Hadoop ecosystem, running its workers as map-only jobs. It relies on the Hadoop Distributed File System (HDFS) for data input and output operations, ensuring compatibility with existing Hadoop infrastructures. Additionally, Giraph employs

Apache ZooKeeper [13] for checkpointing, coordination, and failure recovery mechanisms, enhancing the robustness and reliability of the framework.

Beyond replicating Pregel's basic functionalities, Giraph introduces several advanced features. These include sharded aggregators, which allow for more efficient aggregation processes, and out-of-core computation, which enables the processing of graphs too large to fit entirely in memory. Giraph also supports master computation, allowing for centralized control logic, and edge-oriented input, facilitating more flexible data ingestion methods.

With its rich feature set and robust architecture, Giraph has attracted a growing community of users and developers worldwide. Its popularity is further evidenced by its adoption by major companies, including Facebook, which uses Giraph to handle and process its enormous datasets [14]. This widespread use underscores Giraph's capabilities as a powerful and scalable graph processing framework.

Giraph employs a vertex-centric programming model similar to Pregel, where each vertex in the graph is uniquely identified by an ID. Each vertex also carries additional information, such as a vertex value, a set of edges with associated edge values, and a collection of messages sent to it during computation.

When processing large graphs using the vertex-centric model, it becomes necessary to partition the graph into smaller segments. This partitioning is handled by a partitioner, which ensures that each partition is interconnected with others through cross-edges. The partitioner is also responsible for distributing these partitions across a cluster of worker machines.

In Giraph, the partitioner determines which partition a vertex belongs to based on its ID. By default, Giraph uses a hash function on the vertex ID for this partitioning. However, the framework also supports custom partitioners tailored to specific graph characteristics or optimization goals. These custom partitioners can be configured to meet the specific requirements of different graph processing tasks and environments.

Sandra et al. [15] propose NPEPE as an innovative high-scalability engine designed to execute NPEP (Nested Parallel Execution Patterns) descriptions using Apache Giraph on Hadoop platforms. Sandra et al. [16] also demonstrate NEPO's advantages by presenting a linear-time solution to a well-known NP-complete optimization problem, specifically the 0/1 Knapsack problem. Goldman et al. [17] introduce an efficient algorithm for solving the integral Knapsack problem on a hypercube. Their approach involves representing the computations of the dynamic programming formulation as a precedence graph, structured like an irregular mesh. Subsequently, we propose a time-optimal scheduling algorithm for computing these irregular meshes on a hypercube.

3. GiraDP

In this section, we introduce a new framework for solving optimization problems that can be addressed using dynamic programming. Dynamic programming splits a problem into sub-problems, saves their results, and then uses these saved results to solve larger problems, eventually computing the final and optimal answers. When the problem is divided into sub-problems, some results may be generated that are not needed in subsequent steps. Therefore, we only need some results from the last steps and can ignore others. Before detailing GiraDP, we will describe its architecture.

3.1 System Architecture

The proposed GiraDP architecture is depicted in ***Figure 1***. It is a distributed system consisting of three layers. The lowest layer comprises machines that form the Hadoop clusters, as Giraph utilizes the existing Hadoop installation. Layer 2 incorporates the Giraph framework for processing large-scale graphs. Giraph and Hadoop are among the most well-known and widely used technologies in big data processing. Their ability to handle large-scale distributed computations makes them a standard choice in both academic research and industry applications**.** While their setup and deployment may introduce some complexity, mastering these tools provides significant advantages in solving complex optimization problems efficiently. The top layer is the organizer layer, which consists of three components: JobManager, ComputeManager, and DataManager.

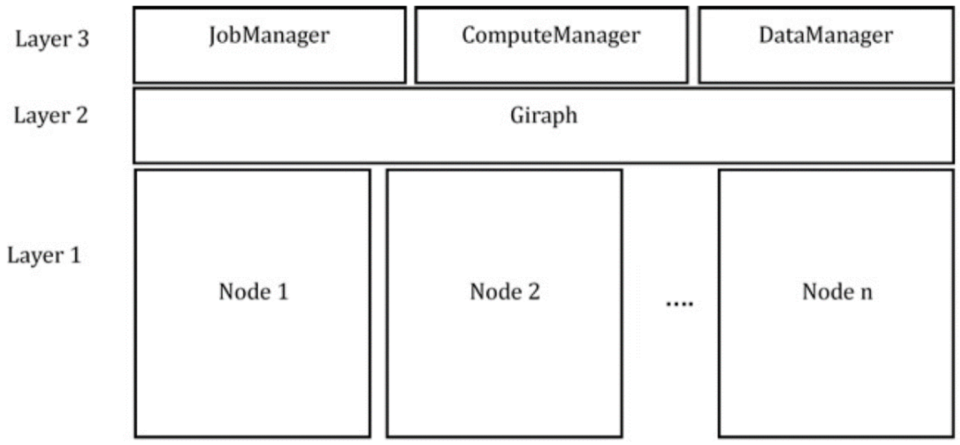

Figure 1. System architecture

First, a user submits the job to JobManager. In GiraDP, the submitted job must include the SetSubProblems function and the IsBasicProblem function. The SetSubProblem function determines which results of sub-problems are required to compute the final answer. In dynamic programming, a problem is divided into smaller subproblems until the simplest form, known as the base case, is reached. The IsBasicProblem function plays a critical role by determining whether a given problem instance has reached this base case. It evaluates the current state of the problem and returns a boolean value indicating if further division is necessary. Once the base case is reached, the division process stops.

Just as in Giraph and Hadoop, where users need to write the required functions(like Map, Reduce or Compute in Giraph), in GiraDP, users must also deploy these functions(SetSubProblem and IsBasicProblem). Note that the SetSubProblems and IsBasicProblem functions are user-defined. After the user submits the job, JobManager divides it into two sub-jobs and extracts the SetSubProblem and IsBasicProblem functions. Using these functions, the ComputeManager creates a new Hadoop job that generates a precedence graph, which serves as input for Giraph. ComputeManager also creates a compute function and sends it to the next layer along with the input graph for Giraph processing. Essentially, it constructs a graph based on the IsBasicFunction and SetSubProblems functions.

The user is required to provide an XML file as input containing detailed information necessary for the system to process. This XML file should include:

- **Dimensions of the problem:** This refers to the size and scope of the problem. For example, some problem in dynamic programing use the array that its dimensions is one or some others use matrix that its dimensions are 2.
- **Dependencies on subproblems:** This outlines how the main problem is related to or dependent on which subproblems.
- **Problem to be solved:** This identifies the specific problem that need to be resolved, guiding the system on where to focus its efforts.

By providing structured information in an XML file, the user ensures the system accurately interprets the problem's requirements and performs the necessary computational tasks. The input data and generated precedence graph are stored in the DataManager. Both ComputeManager and Giraph can independently access the required input data from it. Once the input graph is prepared in Layer 2, Giraph executes the job to solve the optimization problem.

3.2 Compute Manager

In this section, we outline the process of constructing a graph to be used as input for the next layer within a dynamic programming context. Dynamic programming (DP) is characterized by a bottom-up approach where sub-problems are solved incrementally, and their solutions are cached for reuse. This method optimizes computation by eliminating redundant calculations.

To effectively model this process, we use a directed acyclic graph (DAG), where each sub-problem is represented as a vertex (node), and dependencies between sub-problems are represented as directed edges. Here is a step-by-step breakdown of the graph construction:

1. **Sub-Problems as Vertices:** Each sub-problem within the dynamic programming algorithm is represented as a vertex in the graph. This visualization helps illustrate the computational relationships among different sub-problems.
2. **Dependencies as Directed Edges:** If a sub-problem A relies on the result of sub-problem B for its computation, we establish a directed edge from vertex B to vertex A. This signifies that B must be computed before A to maintain correct dependencies.
3. **Constructing the Graph:** Through analysis of these dependencies, we construct a directed acyclic graph (DAG). A DAG ensures no cycles exist, thereby establishing a clear computation order from initial sub-problems to the final problem.

Consider Matrix C shown in ***Figure 2***. This matrix is used to illustrate the dependencies among sub-problems in a dynamic programming algorithm.

$$C = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & a & b & c \\ 6 & d & e & f \\ 7 & g & h & i \end{bmatrix}$$

Figure 2. Example of a matrix as required data in the dynamic programming

Assume we have an optimization problem where each element in the matrix is computed based on the sum of specific elements from the previous rows and columns. For instance:

To compute C11, we need the values of C01 and C10. To compute C33 (the final result in the matrix), we need the values of h and f. ‘a’ is computed from C01 and C10. ‘b’ is computed from C02 and C11. ‘c’ is computed from C03 and C12. ‘d’ is computed from C11 and C20. ‘e’ is computed from C12 and C21. ‘f’ is computed from C13 and C22. ‘g’ is computed from C30 and C21. ‘h’ is computed from C31 and C22. ‘i' is computed from C23 and C32. This creates a chain of dependencies that can be represented as a graph. By following the dependencies, we can construct the DAG showing in ***Figure 3***:

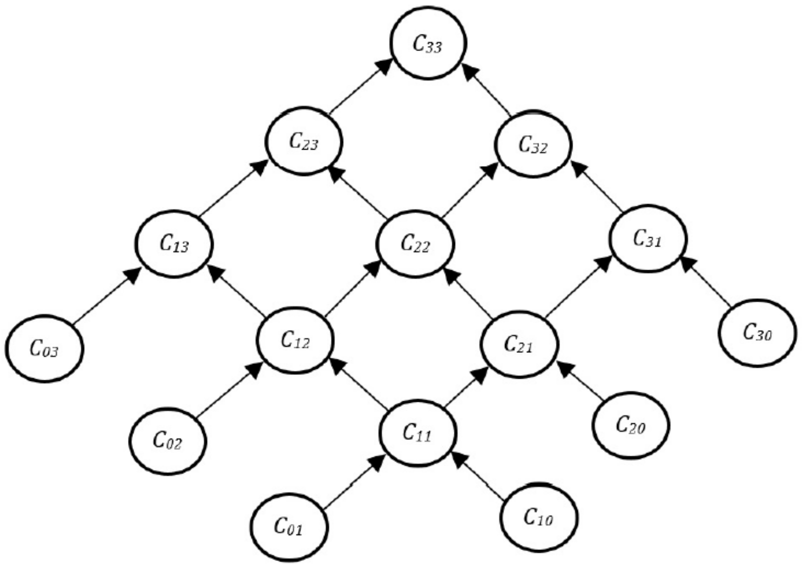

Figure 3. Directed Acyclic Graph (DAG) representing dependencies in the dynamic programming matrix

In this example, we demonstrate how a matrix can be utilized as input data for the subsequent layer in dynamic programming. By representing sub-problems and their dependencies as a Directed Acyclic Graph (DAG), we effectively manage the computation process using Giraph. This method ensures a structured and organized execution of all required calculations.

Algorithm 1 illustrates the creation of a precedence graph. The process starts with the submission of a job to the JobManager. Key functions, SetSubProblems() and IsBasicProblem(), are extracted from the job to identify and manage sub-problems (line 2). The target problem, specified in an XML file, is extracted next and used to initialize the root vertex of a graph.

Algorithm 1 Creating precedence graph

```
1: Submit a job to jobManage;
2: Extract functions SetSubProblems() and IsBasicProblem() from job.
3: Extract target problem from XML file.
4: Create vertex V as root for target problem.
5: ProcessVertex(V)
 function $ProcessVertex(vertex)$
     if $IsBasicProblem(vertex) is false$ then
         SubProblems = SetSubProblems(vertex)
         for each subProblem in SubProblems: do
             Add subProblem as child of vertex
             ProcessVertex(subProblem)
         end for
     end if
 end function
```

The algorithm centers around the ProcessVertex() function (line 5), which handles the processing of each vertex in the graph. The root vertex, symbolizing the target problem, undergoes examination to ascertain if it qualifies as a basic problem using the IsBasicProblem() function. If the root is not basic, the SetSubProblems() function is utilized to identify its sub-problems. Each identified sub-problem is subsequently incorporated as a child vertex in the graph, thus establishing a hierarchical structure. Note that Algorithm 1 is executed as a Hadoop job on a Hadoop cluster to optimize system utilization.

This recursive process is applied to each sub-problem, ensuring that every vertex in the graph undergoes processing. Sub-problems are continually decomposed and integrated into the graph until all vertices are identified as basic problems. This graph-based approach enables organized and efficient management of the computation process, facilitating the resolution of complex problems using dynamic programming techniques. The algorithm guarantees accurate representation and handling of all computations and dependencies, ultimately leading to the solution of the target problem.

4. Performance evaluation

In this section, we conduct experiments to evaluate the effectiveness of the GiraDP in solving optimization problem when the data input is too large. We evaluate the performance of GiraDP using a testbed and compared it to the MPI and serial approaches. Here, we use one applications, knapsack 0/1, with different capacities and items. The 0/1 Knapsack problem is a classic optimization problem that can be solved using dynamic programming. It involves a set of items, each with a weight and a value, and the objective is to determine the most valuable combination of items that can be included in a knapsack of fixed capacity. The term "0/1" indicates that each item can either be included (1) or excluded (0) from the knapsack, with no partial selections allowed. The goal is to maximize the total value V of the items in the knapsack such that the total weight W$'$ does not exceed the capacity W.

The dynamic programming approach to solving the 0/1 Knapsack problem involves constructing a 2D table K where K[i][w] represents the maximum value that can be achieved with the first i items and a knapsack capacity of w. This dynamic programming approach ensures that the optimal solution is found efficiently, with a time complexity of O(n × W). If $W = 2^n$ then the serial approach is not efficient for large number of n. The parallel algorithm using MPI involves dividing the problem into smaller subproblems, distributing them among multiple processors, and combining the results. When dealing with a very high number of items in the 0/1 Knapsack problem, managing a MPI cluster can indeed become challenging.

We ran the 0/1 Knapsack problem on a cluster consisting of 4 nodes. ***Table 1*** shows the specifications of our cluster.

Table 1. Specifications of the Cluster

	CoreNumber	Memory	Storage
Master	6	16 G	120 G
Worker 1	2	4 G	80 G
Worker 2	2	4 G	80 G
Worker 3	2	4 G	80 G

First, we ran the 0/1 Knapsack problem with 20 items and varying capacities using a serial algorithm. The serial algorithm, which uses dynamic programming, was executed on the Master node. Subsequently, we ran the Knapsack problem on the cluster using MPI. Finally, GiraDP was used to solve the Knapsack problem. ***Figure 4*** shows the time taken to solve the Knapsack problem using these different methods.

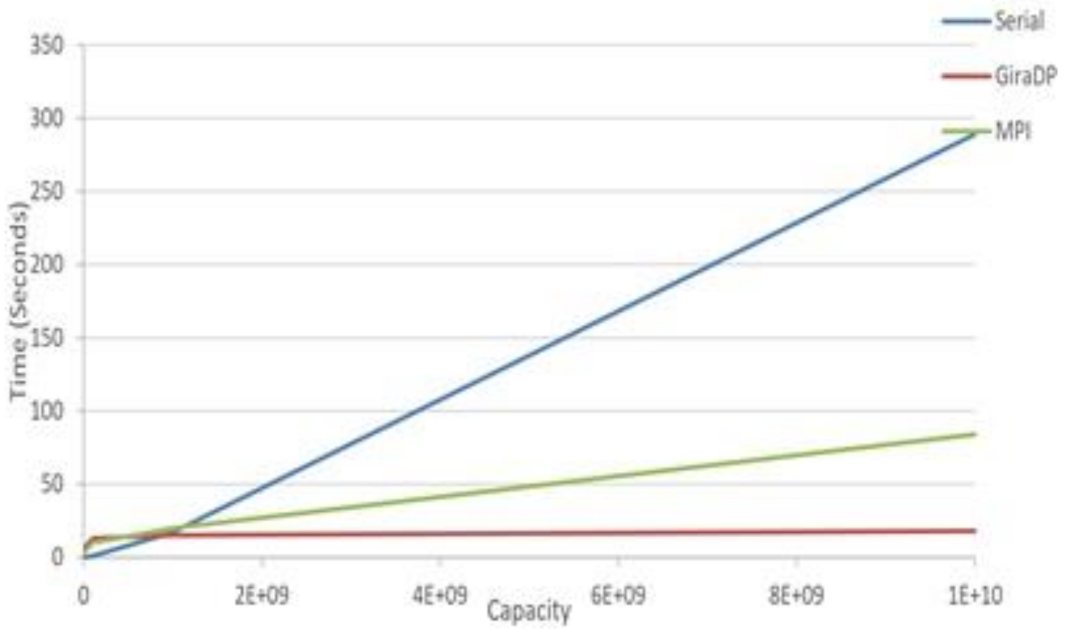

Figure 4. run time for 20 items in knapsack 0/1

As shown in ***Figure 4***, for low capacities, the serial algorithm has the best runtime compared to the others. However, for higher capacities, GiraDP demonstrates the best efficiency, and MPI also outperforms the serial algorithm. The serial algorithm is not a good approach for solving the Knapsack problem at higher capacities. In the next step, we ran the Knapsack problem using various approaches with 40 and 60 items. In the next step, we ran the Knapsack problem using various approaches with 40 and 60 items. ***Figures 5 and 6*** illustrate the efficiency of these approaches. Specifically, ***Figure 5*** presents the runtime comparisons for the different algorithms when handling 40 items, while ***Figure 6*** provides a similar comparison for the case with 60 items. These figures demonstrate how the efficiency of each approach scales with the number of items, highlighting the strengths and weaknesses of the serial algorithm, MPI-based solution, and GiraDP across different problem sizes.

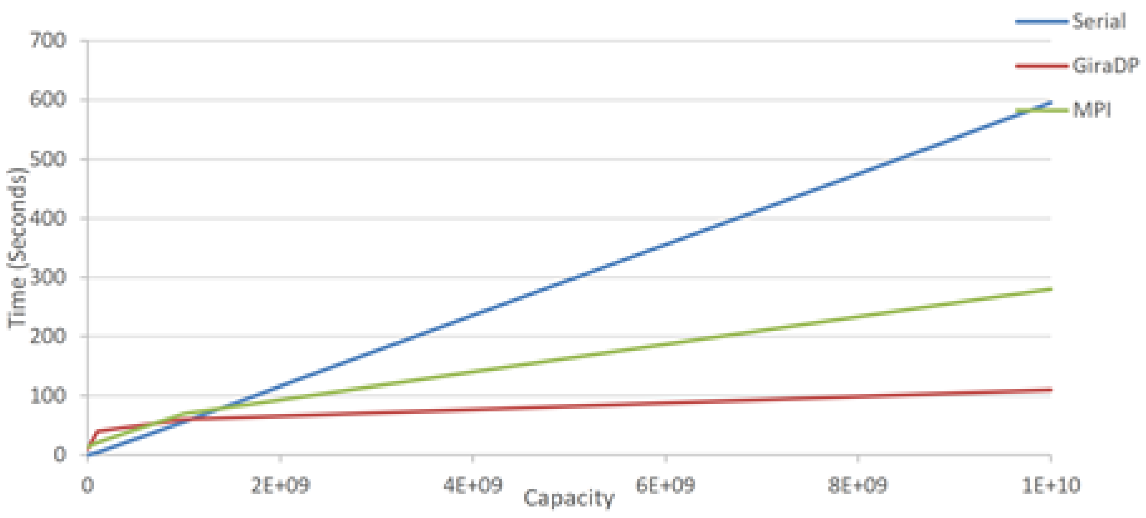

Figure 5. run time for 40 items in knapsack 0/1

As shown in ***Figures 5 and 6***, the serial algorithm loses its efficiency as the number of items increases. When dealing with high capacities, the serial algorithm is unable to solve the Knapsack problem, often resulting in a heap space error due to insufficient memory. This demonstrates the limitations of the serial approach in handling large-scale problems. In contrast, GiraDP shows significantly better efficiency when dealing with a large number of items and high capacities. While GiraDP's performance is not as efficient at low capacities compared to the serial algorithm, its efficiency markedly improves as the problem size increases. This scalability makes GiraDP a superior choice for solving larger instances of the Knapsack problem. Moreover, the MPI-based approach also performs better than the serial algorithm at higher capacities, though it does not match the efficiency of GiraDP. These results highlight the advantages of distributed computing approaches like MPI and GiraDP over traditional serial algorithms, especially for large-scale optimization problems. GiraDP may not be as efficient as serial algorithms for smaller problems and low capacities. This is a well-known trade-off in distributed computing**,** where parallelization overhead can outweigh the benefits for smaller-scale problems.

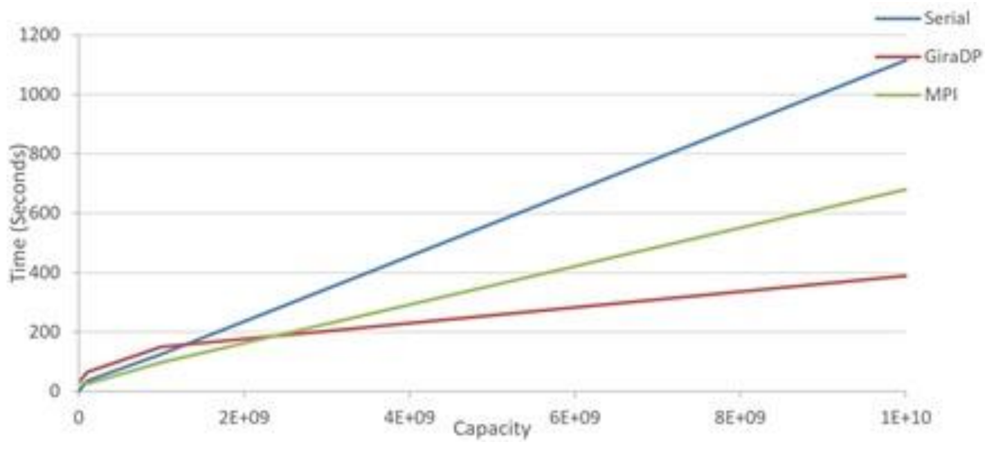

Figure. 6. run time for 60 items in knapsack 0/1

5. Conclusion

In this study, we compared the performance of different approaches to solving the 0/1 Knapsack problem, including a serial algorithm, an MPI-based parallel algorithm, and GiraDP. Our experiments revealed significant differences in efficiency and scalability among these methods. For smaller problem sizes and low capacities, the serial algorithm demonstrated the best runtime performance. However, as the number of items and the capacity increased, the serial algorithm's efficiency drastically declined, ultimately resulting in heap space errors and rendering it unsuitable for large-scale problems. The MPI-based parallel algorithm showed improved performance over the serial algorithm for larger capacities, benefiting from the distribution of computational workload across multiple nodes. However, it still could not match the efficiency of GiraDP. GiraDP exhibited superior scalability and efficiency when handling large problem sizes and high capacities. Although its performance at lower capacities was not as efficient as the serial algorithm, GiraDP's efficiency significantly improved with increasing capacity and the number of items, making it the best approach for large-scale instances of the Knapsack problem. These findings underscore the importance of selecting appropriate algorithms and computational approaches based on the specific problem size and complexity. Distributed computing approaches, such as GiraDP and MPI, offer substantial advantages for large-scale optimization problems, where traditional serial algorithms fall short.

References

[1] M.R. Garey, D.S. Johnson. (1979). Computers and Intractability: A Guide to the Theory of NPcompleteness, W.H. Freeman andCompany, San Francisco.

[2] Itzkovitz, A., and Schuster, A. (1999). Distributed shared memory: Bridging the granularity gap. Technion-Israrl Institute of Technology, Faculty of Computer Science.

[3] B. Murdock. (2014). Learning in a distributed memory model, in: Current issues in cognitive processes: The Tulane Flowerree symposium on cognition, pp. 69–106

[4] M. Redekopp, Y. Simmhan and V. K. Prasan. (2013). "Optimizations and Analysis of BSP Graph Processing Models on Public Clouds", in Proceedings of the IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS'13), Boston, MA, USA

[5] "Apache Giraph", [Online]. Available: http://giraph.apache.org/

[6] "GraphX", [Online]. Available: http://spark.apache.org/graphx/.

[7] J. Gonzalez, Y. Low, A. Kyrola, C. Guestrin, D. Bickson, and M. Hellerstein. (2012). Distributed GraphLab: A Framework for Machine Learning in the Cloud. Proc. the VLDB Endowment, 5(8):716-727.

[8] Committee on the Analysis of Massive, Committee on Applied and Theoretical Statistics, Board on Mathematical Sciences and Their Applications, Division on Engineering and Physical Sciences and National Research Council, Frontiers in Massive Data Analysis, The National Academies Press, 2013.

[9] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski. (2010). Pregel: A System for Large-Scale Graph Processing. Proceedings of The 2010 ACM SIGMOD International Conference on Management of Data.

[10] L. G. Valiant. (1990) A bridging model for parallel computation. Communications of the ACM, vol. 33, no. 8, pp. 103-111.

[11] L. Cao, GoldenOrb. (2011). [Online]. Available: https://github.com/jzachr/goldenorb. [Accessed 25 July 2015]

[12] "Apache Hama", [Online], Available: https://hama.apache.org/

[13] "Apache ZooKeeper" [Online]. Available: https://zookeeper.apache.org/

[14] A, Ching, (2013). Scaling Apache Giraph to a Trillion Edges. Availavle: https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-atrillionedges/ 10151617006153920.

[15] Gomez Canaval, S., Ordozgoiti Rubio, B., Mozo, A. (2015). NPEPE: Massive Natural Computing Engine for Optimally Solving NP-complete Problems in Big Data Scenarios. In: Morzy, T., Valduriez, P., Bellatreche, L. (eds) New Trends in Databases and Information Systems. ADBIS 2015. Communications in Computer and Information Science, vol 539. Springer, Cham.

[16] Sandra Gomez Canaval, Jose Ramon Sanchez Couso, Meritxell Vinyals. (2016). Solving optimization problems by using networks of evolutionary processors with quantitative filtering, Journal of Computational Science, Volume 16.

[17] A. Goldman, D. Trystram. (2004). An efficient parallel algorithm for solving the Knapsack problem on hypercubes, Journal of Parallel and Distributed Computing, Volume 64, Issue 11.