Performance Analysis of Parallel Algorithms

Pruthvi N. Shetty
School of Informatics and Computing
Indiana University
Bloomington, Indiana 47405
Email: shettypr@indiana.edu

Sujith Shivaprakash
School of Informatics and Computing
Indiana University
Bloomington, Indiana 47405
Email: sujishiv@indiana.edu

Abstract—In this paper, we provide a qualitative and quantitative analysis of the performance of parallel algorithms on modern multi-core hardware. We attempt to show a comparative study of the performances of algorithms (traditionally perceived as sequential in nature) in a parallel environment, using the Message Passing Interface (MPI) based on Amdahl’s Law. First, we study sorting algorithms. Sorting is a fundamental problem in computer science, and one where there is a limit on the efficiency of algorithms that exist. In theory it contains a large amount of parallelism and should not be difficult to accelerate sorting of very large datasets on modern architectures. Unfortunately, most serial sorting algorithms do not lend themselves to easy parallelization, especially in a distributed memory system such as we might use with MPI. While initial results show a promising speedup for sorting algorithms, owing to inter-process communication latency, we see an slower run-time, overall with increased number of processors.

I. INTRODUCTION

This paper is being submitted jointly as a part of our final project for the CSCI B503 ‘Analysis & Design of Algorithms’ class by Prof. Andy Somogyi, School of Informatics & Computing, Indiana University, Bloomington, and is intended to serve as a guide to the inferences we have drawn from our comparative study of parallel algorithms, complementing the data we have been able to gather as results of this project and the corresponding visualizations, which help us gain a better understanding of the behavior of algorithms in a parallel environment.

May 04, 2016

A. Concepts & Algorithms

In this paper, we have implemented the Bubble Sort, Quick Sort, K-means and Parallel Search algorithms using Message Passing Interface (MPI) approach.

1) Amdahl’s Law: In computer architecture, Amdahl’s law (or Amdahl’s argument[1]) gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. It is named after computer scientist Gene Amdahl, and was presented at the AFIPS Spring Joint Computer Conference in 1967.

Amdahl’s law can be formulated the following way:

\[ S_{\text{latency}}(s) = \frac{1}{1 - p + p/s} \]

where:

- \( S_{\text{latency}} \) is the theoretical speedup in latency of the execution of the whole task;
- \( s \) is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- \( p \) is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

The figure shows evolution according to Amdahl’s law of the theoretical speedup in latency of the execution of a program in function of the number of processors executing it, for different values of \( p \). The speedup is limited by the serial part of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times.

2) Message Passing Interface: Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public
domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications. MPI provides a rich range of abilities. The following concepts help in understanding and providing context for all of those abilities and help the programmer to decide what functionality to use in their application programs.

3) Communicator: Communicator objects connect groups of processes in the MPI session. Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology. MPI also has explicit groups, but these are mainly good for organizing and reorganizing groups of processes before another communicator is made. MPI understands single group intracommunicator operations, and bilateral intercommunicator communication. In MPI 1, single group operations are most prevalent. Bilateral operations mostly appear in MPI 2 where they include collective communication and dynamic in process management. Communicators can be partitioned using several MPI commands. These commands include MPI_COMM_SPLIT, where each process joins one of several colored sub-communicators by declaring itself to have that color.

4) Point-to-point basics: A number of important MPI functions involve communication between two specific processes. A popular example is MPI_Send, which allows one specified process to send a message to a second specified process. Point-to-point operations, as these are called, are particularly useful in patterned or irregular communication, for example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other processors between calculation steps, or a master-slave architecture in which the master sends new task data to a slave whenever the prior task is completed. MPI-1 specifies mechanisms for both blocking and non-blocking point-to-point communication mechanisms, as well as the so-called 'ready-send' mechanism whereby a send request can be made only when the matching receive request has already been made.

5) Collective basics: Collective functions involve communication among all processes in a process group (which can mean the entire process pool or a program-defined subset). A typical function is the MPI_Bcast call (short for "broadcast"). This function takes data from one node and sends it to all processes in the process group. A reverse operation is the MPI_Reduce call, which takes data from all processes in a group, performs an operation (such as summing), and stores the results on one node. MPI_Reduce is often useful at the start or end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result. Other operations perform more sophisticated tasks, such as MPI_Alltoall which rearranges n items of data such that the nth node gets the nth item of data from each.

6) Bubble Sort: Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.

7) Quick Sort: Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959, with his work published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort. Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

1) an element, called a pivot, from the array.
2) Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way).
3) After this partitioning, the pivot is in its final position. This is called the partition operation.
4) Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which never need to be sorted.

8) K-Means: K-means is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centroids, one for each cluster. These centroids should be placed in a cunning way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the
same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more. Finally, this algorithm aims at minimizing an objective function, in this case a squared error function. The objective function:

\[ J = \sum_{j=1}^{k} \sum_{i=1}^{n} ||x_i - c_j||^2 \]

where \( ||x_i - c_j||^2 \) is a chosen distance measure between a data point and the cluster centre, is an indicator of the distance of the n data points from their respective cluster centres. The Algorithm consists of the following steps:

1) Place K points into the space represented by the objects that are being clustered. These points represent initial group centroids.

2) Assign each object to the group that has the closest centroid.

3) When all objects have been assigned, recalculate the positions of the K centroids.

4) Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.

9) Parallel Search: SEARCH_MPI is a C program which searches integers between A and B for a value J such that \( F(J) = C \), using the MPI parallel programming environment.

B. Methodology

1) MPI Implementation: We are implementing MPI code using Master/slave communications where the first thing we have calculates the size of chunks equal the length of the array divided by the number of processors. After that the master will send the data to the slaves and each slave will do a bubble sort on own data. At the end of this step the slave receives the result to master. The final step merges all the chunks and prints the results. Figure 1 described the proposed method for sorting.

2) Parallel Bubble Sort: One of the fundamental problems of computer science is ordering a list of items. There are a lot of solutions for this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort, but others, like quick sort, are extremely complicated but produce lightning-fast results. The sequential version of the bubble sort algorithm is considered to be the most inefficient sorting method in common usage. In this assignment we want to prove that how the parallel bubble sort algorithm is used to sort the text file parallel and will show that it may or may not better than the sequential sorting algorithm. The old complexity of the bubble sort algorithm was \( O(n^2) \), but now we are using the complexity for bubble sort algorithm \( n(n-1)/2 \). Algorithm 1 (in chapter 1) shows the code for the bubble sort algorithm. As usually in parallelism we can decompose the data or functions or sometimes both, but after we are studying the bubble sort we reached to the point, that it is more suitable for data decomposition. So we suggest during the implementation of MPI is to assign each vector to individual process and during the implementation of MPI we divide each vector to different chunks and assign each chunk to individual process, as a result we merge all the executed chunks for the solution of the problem. A way to implement the Bubble Sort in parallel is to divide the list of data (more or less) equally between the N-1 nodes to (N-1) of an N nodes parallel machine, keeping node 0 to administer the calculation. Each node 1 to (N-1) can then sort its partial list and send it back to node 0 for a final global merge.

3) Parallel Quicksort Implementation: In this attempt, the main idea was to implement a parallelized Quicksort to run on a multi-core environment and conduct a performance evaluation. This parallelization is obtained by using MPI API functionalities to share the sorting data set among multi processes. In its simplest form, the parallel implementation of the Quicksort Algorithm can be similar to that of the Bubble Sort. Initially, perform an initial partitioning of data until all the available processes were given a subset to sort sequentially. Next, Sort the received data set by each process in parallel. Finally, gather all the data corresponding to the exact partitioned offsets without performing any merging. A more efficient implementation could take advantage of the relative ordered ranges of the left-right sub-lists in the algorithm. The merging step would be simplified. But the number of nodes would have to be the master node plus a power-of-two worker nodes.

4) Parallel K-Means Implementation: Consider N data points each of it is vector and P processors.

1) Assign N/P data points to each processor.

2) Node 0 randomly choose K points and assigns them as cluster means and broadcast.

3) In each processor for each data point find membership using the cluster mean.

4) Recalculate local means for each cluster in each processor.

5) Globally broadcast all local means for each processor find the global mean.

6) Go to step (3) and repeat until the number of iterations > 10000 or number of points where membership has changed is less than 0.1%.

5) Parallel-Search Implementation: Parallel Search, also known as Multithreaded Search or SMP Search, is a way to increase search speed by using additional processors. This topic that has been gaining popularity recently with multiprocessor computers becoming widely available. Here, we use the algorithm to search for an integer between lower bound, A and upper bound, B for a value J such that \( F(J) = C \), a constant, using the MPI parallel programming environment.
C. Results

1) Parallel-Bubble Sort: We analyze the parallel bubble sort algorithm with varying number of processors, ranging from 1 to 64. From the given graph given below, we can see that the performance initially improves while we increase the number of nodes from 1 to 16, owing to higher computing power and then drops once the numbers of processors goes beyond 16. This is attributed to the fact that the interprocess communication adds to the overhead of the total run time of the algorithm.

2) Parallel Quicksort: A similar situation occurs in case of quicksort. We can see that a increased number of processors initially result in a better performance, but once the number of processors increases above 32, the performance degrades dramatically, the overhead of interprocess communication taking over the performance of the algorithm.

3) Parallel K-means: In case of K-means, the performance tends to increase initially, before a sharp decline beyond 16 processors. This is due to the latency caused by the inter-process communication. Likewise, we can see the trendline plotted to show a linear degradation in performance with the increase in processing power.

4) Parallel-Search: The study proves that this is an embarrassingly parallel workload or problem (also called perfectly parallel or pleasingly parallel). This is often the case where there is little or no dependency, or need for communication between those parallel tasks, or for results between them. Thus we see a increased performance on the part of the algorithm with increased number of nodes. We have tested the algorithm over a range of 1-256 nodes with a constant increase in run-time.

II. Conclusion

In case of Sorting Algorithms, this study can be seen as a classic example of the case where mere increase in the number of processors does not necessarily translate into faster run-times. We see a similar trend with the parallel K-means implementation. However, with our forth implementation, we witness an embarassingly parallel algorithm with Parallel Search where we find that with the increase in the number of nodes directly results in a better performance. This can be viewed as an example to show performance gain being directly proportional to computing power, without depending on latency caused by communication overhead.

We can conclude from this study that we need to consider the inter-process communication latency which might affect the performance of the algorithm, proving to be an overhead. An increase in number of computing nodes does not always
guarantee a better performance gain, and in fact might degrade the algorithm’s performance. There is a need to find the right trade-off between computing power and communication overhead for optimal performance gain. Studies in this direction can be used in making decisions about resource utilization.

ACKNOWLEDGMENT

We wish to express our sincere gratitude to Prof. Andy Somogyi for encouraging us to take up this project and implement it. We sincerely thank him for his immense support and feedback provided throughout the duration of this project.

REFERENCES

[5] Running MPI jobs on Karst at IU