CS420 Project: Accelerating CKY Algorithm using MPI

Adithya Renduchintala, Ankit Garg

May 13, 2015

Abstract

CKY[1] (also called CYK algorithm) is a popular algorithm for parsing context free grammar in the domain of Natural Language Processing. We developed a parallel version of the popular CKY parsing algorithm using Open MPI[2]. The CKY algorithm is a bottom up tree based algorithm that builds a parse tree from an input sentence and algorithm fits nicely into a message passing paradigm which motivated using Open MPI. To our knowledge an MPI based implementation of CKY algorithm doesn't exists though it has been parallelized using other frameworks. Our experiments show large improvements in runtime of CKY algorithm which was consistent with our expectations.

1 Introduction

CKY parsing algorithm works with grammar which is in Chomsky Normal Form. Such grammar contains 2 sets of rewrite-rules, *Binary Rules* permit two Non-terminal symbols to combine into a new Non-terminal symbol and *Unary Rules* that convert a Terminal symbol into a Non-terminal symbol. A special non terminal symbol S is made the root symbol. If the algorithms ends up with S at the root of the tree then a parse is obtained. The algorithm builds sub-trees for every possible sub-string of the input sentence in a bottom up fashion. In the first stage it builds sub-trees for sub-strings of length 1 (using Unary rules), then expands to sub-strings of length 2 using binary rules applied on the sub-trees generated by the previous stage. Eventually reaching the final stage where a tree is built for the entire input string.The CKY algorithm has complexity of $O(n^3|G|)$, where n is the input sentence length and $|G|$ is the number of rules in the grammar.

MPI is a standard library which is used to send messages between different multiple processes. These processes could be running on same machine or on a cluster of distributed machines having network connectivity. Underlying assumption in the implementation is that multiple processes do not share the memory address space with other processes and as a result changes made by one process are not visible to other processes and do not impact them. We need to be explicit in sending message to target process if we want to make it aware of the change.

[1] `http://en.wikipedia.org/wiki/CYK_algorithm`

[2] `http://www.open-mpi.org/`

Book	the	flight	through	Houston
S, VP, Verb, Nominal, Noun [0, 1]	[0, 2]	S, VP, X2 [0, 3]	[0, 4]	S_1, VP, X2, S_2, VP, S_3 [0, 5]
	Det [1, 2]	NP [1, 3]	[1, 4]	NP [1, 5]
		Nominal, Noun [2, 3]	[2, 4]	Nominal [2, 5]
			Prep [3, 4]	PP [3, 5]
				NP, Proper-Noun [4, 5]

Figure 1: Structure of CKY matrix/chart.

2 Implementation

Figure 1[3] gives an example of CKY parse chart/matrix. We start by filling the cells lying on the diagonal and then we move away from diagonal towards top-right. In our implementation, we assigned each cell to a process and that particular process was responsible for receiving all messages required to process the chart and sending results up the chain.

The main challenge of this project was to design a message passing discipline amongst all processes so that there is no deadlock situation and time spent in waiting for a message is minimized. To ensure maximum independence between processes, we started traversing diagonally and assigned each cell to next process and starting at first process once all process have been assigned at-least one cell [in case number of processes are more than number of cells, we are afraid that those processes will not be used as this problem is scalable only upto the number of cells in chart shown above]. Each process starts by processing the cells closest to the diagonal.

One implementation details worth mentioning is that even though at each cell it is possible to have more than one rewrite rule, we are not emitting them one by one. All rules at given cells are combined and encoded into a string, which is then sent to relevant target nodes. We chose to send messages in bulk for two reasons :

1. To avoid overhead costs associated with sending multiple messages (potentially small messages)
2. Because our problem statement involves giving best parse, at each cell, we are removing redundant parses with most probable parse. Receiving messages one by one would require us to maintain a state of all messages processed for that cell, which again would affect the performance adversely.

As it is not possible to cover all details of code in constrained space, we have annotated our main class with appropriate comments and we would request reader to refer to the code if some details are not clear.

[3] `http://www.inf.ed.ac.uk/teaching/courses/inf2a/slides/2014_inf2a_L18_slides.pdf`

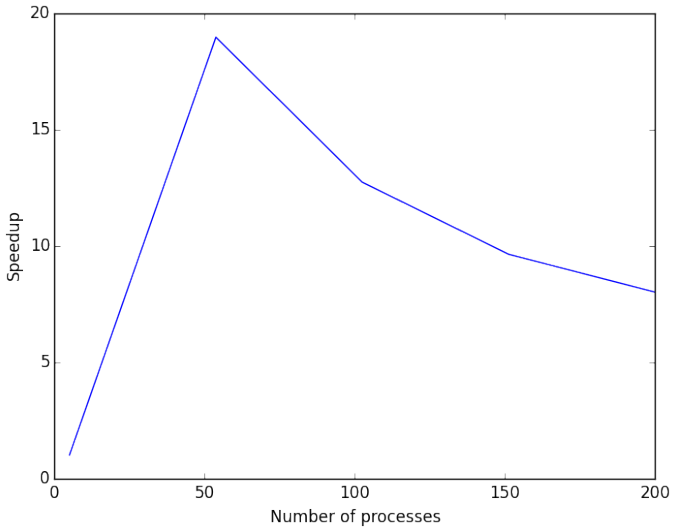

Figure 2: Speedup observed wrt. number of processes.

3 Experiments and results

We ran multiple pipelines to measure speedup obtained as we increased the number of processes to be used (refer to figure 2). Experiments were run on *gradx* cluster of CS department. *gradx* cluster has 24 cores each of which supports hyper-threading.

Speedup was observer until we increased the number of processes to 58, beyond which we saw a decline in performance. Also speedup observed was sub-linear. Our hypothesis for observing speedup till 60 processes on a 24 core machine is that each processor is capable of hyper-threading so effectively we have 48 virtual processors. Also as considerable I/O is involved in our implementation, we think while one process is waiting on I/O, other process is able to progress. Impact of this will diminish as we use much large number of processes.

We varied method to send message between MPI_Ssend and MPI_Send. We noticed that MPI_Ssend was more than an order of magnitude slower than MPI_send when large number of processes were involved, which was consistent with the concepts learned in class. However successfully running our implementation with MPI_Ssend method ensured us that our implementation is deadlock free.

4 Conclusions and other remarks

We have been able to successfully demostrate that CKY algorithm can be highly parallelized using MPI. Our algorithm is highly parallelized and imposes no restriction on number of processes which can be used.

We believe that this implementation can be further optimized if problem statement is modified to just check if a valid parse exists instead of returning best parse. We believe that in such a

scenerio, each process can share its data with other processes in smaller batch so that once any valid parse is found, we can discard remaining messages.

5 who did what

Both Aditya and Ankit contributed in deciding the project problem. While Aditya worked on parsing grammar, Ankit worked on a naive serial implementation of CKY. Both Aditya and Ankit brainstormed to decide message passing discipline and wrote parts of it. Experiments and report writing was done by both of us in parts.