Parallel Web Server for Program Verifiers

14 July 2009

Introduction

A parallel web server is a common interface for program verifiers on a network. The webserver has one or many provers running parallely on the server itself and/or on remote machines. These provers are abstracted from the clients. Clients establish connection with the web server and send the required formula along with the theorem provers to be used to prove them. The webserver can also be given a priority with each of the incoming jobs, and it pushes the jobs into a common priority queue. The server identifies idle servers and assigns them jobs by extracting jobs from the queue one by one.

Web Server Architecture

The web server acts as a master and the provers underneath it act as slaves. Initially the webserver is given the addresses and the ports on which to start prover servers. (A machine can be specified more than once, but must give the same port) After starting a prover on each of the cluster machines, it then connects to these provers to actually start the prover processing loops. The idle ones wait for jobs to be assigned to them by the web server and the ones that are busy flag themselves as idle once they finish their task.
The server does the following main tasks:

- listen for incoming connection from new client
- listen to existing clients for jobs and push these jobs into the priority queue
- assign jobs in the priority queue to idle provers
- listen for results to comeback from provers and send these results back to the appropriate client

Modifications made in the Prove Module

The old prove module worked as follows , it would receive a job and a list of theorem provers to be used. The result would have to be sent back serially.

When more than one theorem prover is used, it would have those many slave processes trying to solve the formula, each one using a different theorem prover. When one of them finsihes the latest job, every prover is buffered with the next job. Ideally one would like to stop all other provers when one of them has finshed the job and all of them to start on the next job, but this would require us to kill the other processes and spawn new ones, which is an expensive operation. Implementing this procedure has the overhead of killing and spawning a larger number of processes. So instead in the modified Prove module when one of the slave provers finishes the latest job, only that slave is given the next job and in the mean time if some other slave finishes some previous job it is given the first job which is not yet solved (formula where a conclusive result is yet to be obtained).

Types of parallelization

Top level parallelism

When using the program verifier on a porgram with many methods, each method can be verified independently of the other. This is also known as coarse parallelism. To solve them parallely a new process is forked for every procedure in which it is verified, and thus exploiting multicore cpu. But running too many processes at a time can be a burden on the cpu, and running very few might not be able to exploit the multi-core fully. We essentially have a trade off between avoiding the swarming of our resources and exploting parallelism to its fullest extent. So we have designed a scheduler that controls the number of procedures that are verified at a time. This number can be given as an input. At best an n-core processor can show speedup upto n times of the sequential case. If we fork more number of processes than the number of cores, then the processor interleaves (time sharing) the processes (Note: Each process can run on a single core , it cannot switch between cores) The above features are implemented in the Paralib modules(Paralib1 and Paralib1v2). They implement the map_para function which is a parallelized alternative to map.

Bottom level parallelism

is explained in modifications in prover module, wherein each formula is attempted by more than one theorem prover.

Advantages of webserver

- If one has several programs to be verified it would be better to do it thru the web server rather than doing sequentially.
- webserver has a pre determined number of provers working for it which makes sure that the cpu resources are not swarmed.

- it acts as a single interface to all clients , so clients dont have to connect to each and every machine on the cluster, they just connect to one webserver

Motivaton for having Priorities

The main issue with parallelizing a number of procedures is that the procedures would be of different sizes and so they take varying amounts of time to solve. The overall time taken will depend on how one chooses to schedule the procedures. A procedure can be scheduled by assigning priorities to its formulae, and the webserver chooses to do those formulae first which have a higher priority.

Strategies to be adopted in assigning priorities

One good way of scheduling the procedures is by using a greedy strategy. Finish the bigger procedures as soon as possible, leaving behind the smaller ones to finish later. This can be done by assigning a higher priority to those formulae that are sent by a bigger procedure.
A way to utilize all cores uniformly would be to assign each formula with priority equal to the number of formulae succeeding it in the procedure which it came from. This makes sure that all the procedures end at almost the same time, thus making sure all the cores are used to the same extent.

Scope for Improvement

- implement a fault tolerating mechanism wherein, if no result is obtained after a pre specified timeout or the results are inconclusive then the prover is once again made to attempt the particular formula
- caching mechanism can be developed to store the results of some formulae which take particularly long time to solve, and if these formulae occur in future runs then use the stored result, instead of solving them.

Experimental Results

Advantage of having a webserver:
when the example program 2-3trees.ss was run without webserver on a 4-core machine with maximum parallelism it took 48 seconds to solve and when the same file was run 4 times using different 4 hips but only a single webserver(with 4 parallel provers) the total time it took was 120 seconds, but if we were to run the 4 of them one after the other we would expect it to take 192 seconds. This shows that the webserver utilizes each core to greater extent in a limited amount of time. The likely reason for this is that there is a time gap between solving of two consecutive formulae from a procedure, which is exploited by the webserver.

The following is the result obtained for running a particular example program with different degrees of top level parallelism on a 4-core cpu

number of procedures running parallel	time taken(seconds)	time taken when procedures are sorted
1	223	NA
2	112.1	112.34
4	70.83	67
8	61.23	58.02
12	59.95	57.56
16	59.27	57.48
all	57.88	57.75