Implementation of Connect4 game using Scala Actor system

Submitted By

Trivikram Bollempalli (1504289)
Email: tbollemp@ucsc.edu

Naga Venkata Sai Indubhaskar Jupudi (1472578)
Email: njupudi@ucsc.edu

School of Engineering
University of California, Santa Cruz
1156 High St, Santa Cruz, CA 95064
2016-2017

Abstract:

Connect Four is a two-player game in which the players are assigned a color at the beginning and they start playing the game by placing (dropping) the colored discs in a board of an arbitrary size (generally a board of size 6x7 is used).

Scala is a general purpose functional programming language which has functionalities like currying, pattern matching, lazy evaluation and also has built in advanced libraries like Scala actor system to implement parallelism[4]. We have used minimax algorithm (formulated for two player zero sum games[2]) with alpha-beta pruning. The main goal of this project is to implement the Connect4 game using Scala actor system and compare the performance between implementation of the game with and without parallelism and the ease of using scala actor system. The game is designed to work for an arbitrary size of the board and for an arbitrary depth of evaluation of minimax algorithm.

Introduction:

Functional programming is a programming paradigm. It treats the computation as the evaluation of mathematical functions and avoids changing-state and mutable data[4]. Scala programming language is one of a kind of functional programming that is object oriented which has built in advanced libraries like Actor system for implementing parallelism and concurrency. Scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a Java virtual machine. Scala has more advantages than java, where it has optional parameters, named parameters, raw strings, and no checked exceptions[4].

Connect4 is a two player game in which players make alternate moves and when a player makes 4 discs of his own color arranged contiguously either vertical, horizontal or diagonally, he wins. In our project a human (player 1) plays against the machine (player 2) where the machine uses minimax algorithm trying to increase the chance of winning by reducing the chances of the opposite player. Minimax algorithm is generally applicable for zero sum two player games in which one of the players is the machine itself. In Connect4, generally the size of the board is 6x7 which makes search space to be exponential. To reduce this, we implement alpha-beta pruning to minimax algorithm to reduce the search space. Here a depth of search space is defined initially, and the minimax algorithm uses recursion to calculate the possible scores at all the places until that depth. So, the difficulty level can be adjusted by adjusting the depth. Here the recursion grows exponentially, depending on depth and implementing parallelism can greatly improve the performance. So, we use scala's actor system.

Scala's actor system is generally designed to implement parallelism by leveraging the underlying CPU resources like multiple cores thereby improving the efficiency of a program. The general workflow of an actor system is there will be a master system which initiates a worker router system to divide the work among the workers depending on the number of cores present

in the CPU and after all the workers finish their job they return to the master where the master aggregates all the results of the workers.

Algorithm:

The main algorithm that is used for the project is minimax algorithm with alpha-beta pruning. It is applicable for two player zero sum games. In minimax approach for Connect4, a machine will make a move by trying out all possible moves for itself and the opponent for next 'x' rounds where x is configurable. While trying all possible moves, machine tries to maximize its revenue and assumes that the opponent will try to minimize the machine's revenue. Alpha-beta pruning is a technique to control the exponential growth of the algorithm by setting limits on maximum and minimum revenues possible and immediately quit the search thereby reducing the search space.

Pseudo code:

The below method takes a move from player and plays a counter move

```
play()
{
        while(true){
                if(player_turn)
                {
                        // Takes input from opponent player and checks whether he won
                        print("Enter a valid column number")
                        int col = takenumber()
                        printboard();
                        if(hasWin(player))
                        {
                                print("You won the game")
                                exit()
                        }
                }
                else
                {
                        // machine plays by calculating the best move known to it
                       makeMove(depth, true, -1);
                       printboard()
                       if(hasWin(machine))
                        {
                                print("I won the Game! What say?")
                                exit()
                        }
                }
                        player_turn = !player_turn
        }
```

```
}
```

This method check whether the board has a win

haswin()

```
{
        //checks for a connect 4 pattern in all 4 directions from the recent move
        checkHorizontal()
        || checkVertical()
        || checkDiagonalRi()
        || checkDiagonalLi()
}
```

This method calculates the move and play accordingly

makeMove(depth, isMaximize, currentmove)

```
{
        If (haswin(machine))
                return MAX_VALUE/(totaldepth- depth);
        else if(hasWin(player))
                return MIN_VALUE/(totaldepth - depth);

        for all moves
        {
                if(depth == 1){
                        //evaluating board favorability for machine
                        tempscore=util.getScore();
                        if(isMaximize)
                                score = maximum(score,tempscore)
                        else
                                score = minimum(score,tempscore)
                }
                if(depth > 1)
                {
                        //making a move and recursively exploring further search space
                        tempscore = makeMove(depth-1, !isMaximize, currentmove);
                //maximize or minimize scores for current moves depending on who is playing
                        if(isMaximize)
                        {
                                score = maximum(score,tempscore)
                                if(depth == totaldepth)
                                {
                                        storecurrentmove();
                                }
```

```
                        }
                        else
                                score = minimum(score,tempscore)

                }
                        undomove();
                }
                return score;
        }
getscore()
{
        //This method returns a score based on the number of discs already connected and free
//space available to make more moves to win;

        calculateverticalscore(connectedcoins, freespaces) +
        calculatehorizontalscore(connectedcoins, freespaces)+
        calculatediagonalLiscore(connectedcoins, freespaces)+
        calculatediagonalRiscore(connectedcoins, freespaces)
}
```

Using Scala Actor system for parallelization:

We can induce parallelism in the connect4 game. While trying to evaluate all possible moves at a given depth, each move is independent of other moves that are occurring at same depth. If we can give a different copy of the board at each depth, we can parallelize those computations. Using this information, we can parallelize the code using scala Actor system. In scala actor system, each actor talk to other through messages as shown in Fig.1. We can assign role of a master to an actor and role of a worker to another set of actors. Each master creates the worker actors and send a message to them using a router, for example, Roundrobin router. The master encapsulates the arguments in a message and send it via router to workers. Workers perform tasks based on the message it received. Once the task is performed, it can send the result to the master again by using messages. The master aggregates the messages received. Now the master needs to send result back to the caller. If the caller is an actor, it can send a message to the caller. Otherwise, scala provides the concept of **Future.** Using futures, we can specify the caller thread that a result will be given to it in future. Now we can block the caller thread if we need the result to proceed further, using Await.result(future, timeout) method provided by scala. The overall workflow between masters and workers is depicted in Figure.1.

Number of workers that can be created can be given as arguments based on the number of cores present in the machine. Scala helps us to clearly separate the number of cores available and number of threads that can be created. Even though number of threads that can be created exceeds the cores, at a given time, number of threads that are executing will be equal to the number of cores given as input.

Design:

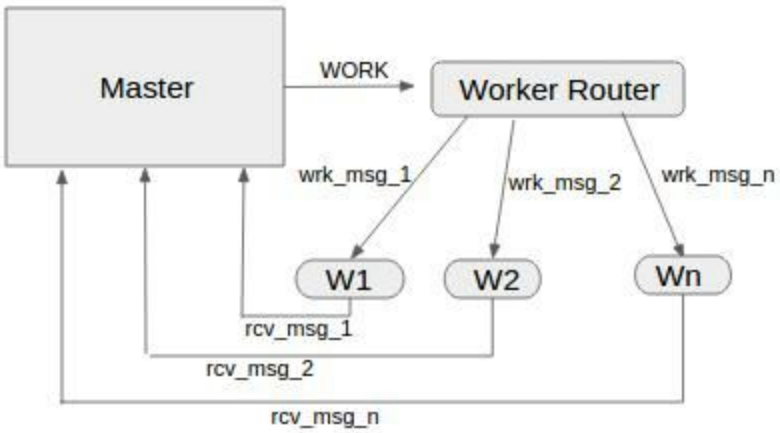


Fig.1. Work flow of Scala Actor system

Pseudo code:

```
 class Master(workers, threadspossible) extends Actor {
  // creating a router
   val workerRouter = context.actorOf(
    Props[Worker].withRouter(RoundRobinRouter(workers)), name = "workerRouter")

//receive method to receive messages from worker actors
   def receive = {
    case Calculate ⇒
     for all threadspossible
     {
        workerRouter ! Work(depth-1, !isMaximize, currentmove)
      }
    case Result(cuurentscore) ⇒
     if (isMaximize) {
      maximizescore();
      if (depth == totaldepth) {
       if (score <= currentscore) {
        Storemove = currentmove
       }
      }
     } else {
      minimizescore();
     }
```

```
      nrOfResults += 1
      //replying the caller with the result when we got replies from all threads created
      if (nrOfResults == allthreadpossible) {
         sender ! score
        context.stop(self)
      }
    }

  }




class Worker extends Actor {

    def calculateoneMove(ismaximize,depth,currentmove) {

    for (i <- 0 until columns) {
      if (depth == 1) {
        val tempscore = getScore()

        if (isMaximize)
         maximizescore()
        else
          minimizescore()
      }
      if (depth > 1) {
        val tempscore = calculateoneMove(!ismaximize,depth-1,currentmove)
        if (isMaximize) {
          maximizescore()
        } else {
         minimizescore()
        }
      }
      undomove()
     }
    }
    score
    }

    //receive method taking messages from master and replying back to the master
    def receive = {
```

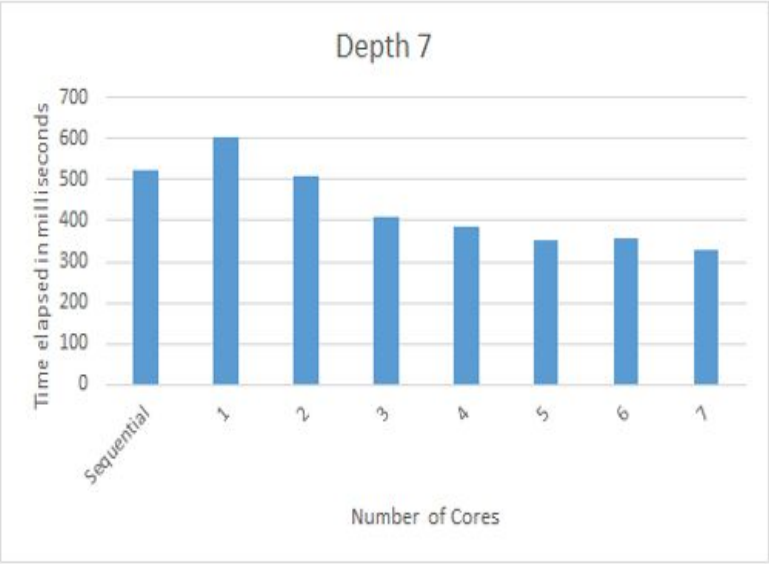
```
    case Work(mat, pos, depth, isMaximize, currRow, currCol, i, bscore, obj) ⇒
      sender ! Result(calculateoneMove()) // perform the work
  }
}
```

Analysis:

Performance analysis:

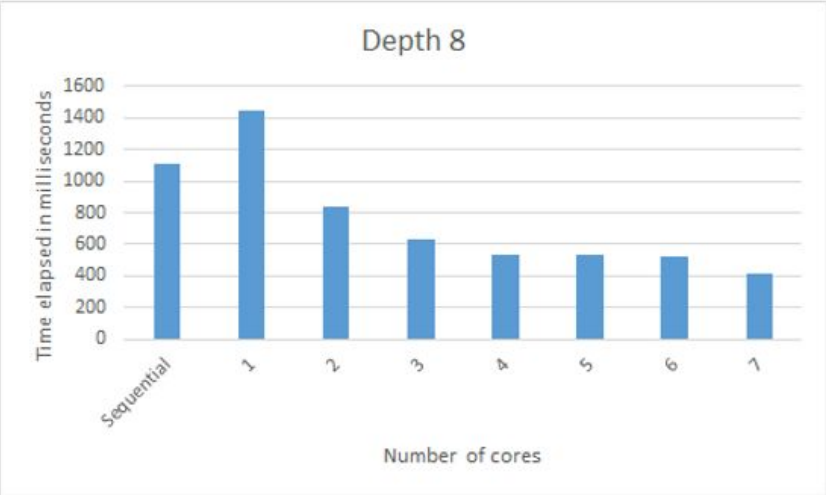
When we started to induce parallelism, we tried to parallelize at each depth. The consequences are drastical. We ended up creating 7^7 threads for a board of size 7*7 with deoth 7 as search space. Obviously, this made the game very slow. We realized our mistake and parallelized only for the first depth and calculating scores for further depths is done sequentially.

The below graphs are drawn by calculating the mean for 6 readings by varying depth (7-10) of search space and number of cores for a board of size 7x7. It is evident that, by implementing parallelism the performance of the program is enhanced. We also calculated the standard deviation of the results that are obtained to measure the skewness of the data. Here for depth 7 as the search space is low when compared to other depths, the performance enhancement is low. For depth 10, as the search space is considerably large, implementing the scala's actor system resulted in a significant performance gain.



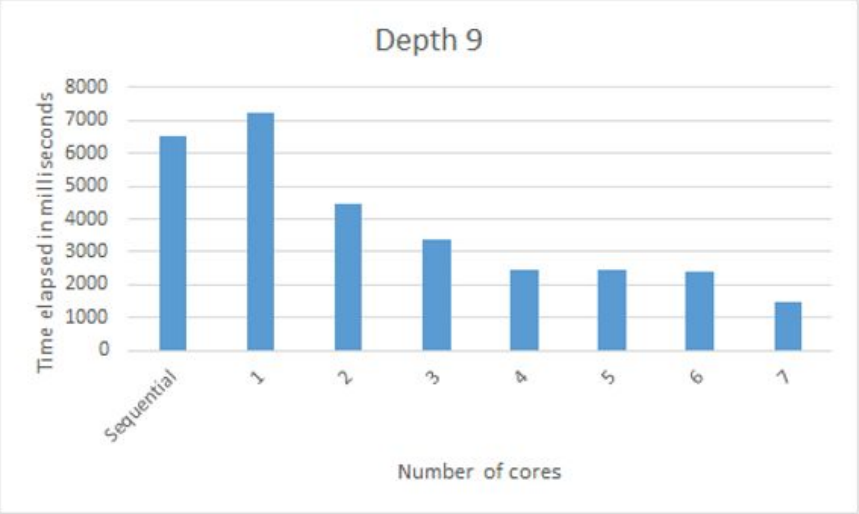
No.of cores	Standard deviation
Sequential	34.24138172
1	35.07333587
2	16.2215563
3	13.1835335
4	13.1497782
5	18.57791401
6	6.817053779
7	4.5

Fig.2 Mean time and standard deviation elapsed in milliseconds for depth 7



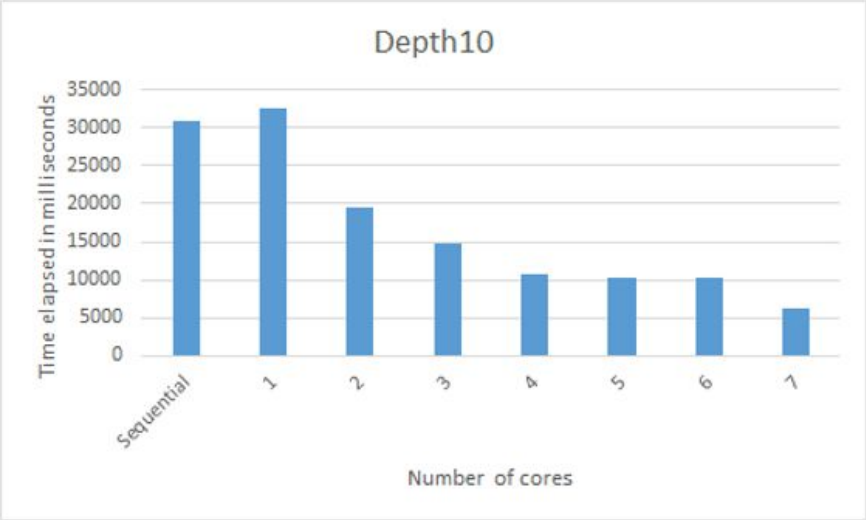
No.of cores	Standard deviation
Sequential	151.9590588
1	302.2686443
2	42.84338403
3	23.62672686
4	15.40202007
5	17.14318783
6	12.23383287
7	7.514800212

Fig.3 Mean time and standard deviation elapsed in milliseconds for depth 8



No.of cores	Standard deviation
Sequential	218.0970732
1	18.4812217
2	121.3026518
3	25.24986249
4	82.44476265
5	101.8381068
6	27.04317536
7	40.36259269

Fig.4 Mean time and standard deviation elapsed in milliseconds for depth 9



No.of cores	Standard deviation
Sequential	1061.097807
1	239.5688256
2	713.8995416
3	176.7295486
4	211.7840226
5	64.3801643
6	110.6386762
7	135.143134

Fig.5 Mean time and standard deviation elapsed in milliseconds for depth 10

Other important observation that can be deduced from these graphs is when number of cores that are used is 1, then its performance is not better than the sequential approach as it is equivalent to as implementing the sequential method with an extra overhead of initialising all the threads. When number of cores are 4,5 or 6, the performance is almost the same. This is because we are initialising 7 threads since the column size of the board is 7. When we use 4 cores, initially 4 threads are evaluated first, then other 3 threads are evaluated. Here the CPU has to evaluate all the threads in 2 stages i.e., evaluating 4 threads first and then 3 other threads later. Sam is the case when we use 5 or 6 cores. But when we use 7 cores all the threads are initialized and evaluated simultaneously in a single stage. So use of 7 cores drastically improves the performance of the game when compared to 4,5 or 6 cores.

So by implementing parallelism using scala actor system , the performance is enhanced noticeably when depth is increased from 7 to 10 as the search space is increased considerably.

Analysis on ease of use of scala for parallelism:

If a yes or no question is asked to say whether scala made life easy while implementing parallelism, the answer is a definite **yes.** Many features of scala Actor system provides an easy to use interfaces to them. Some of its features worth mentioning are:

Cores and Threads are configurable: Number of workers that can be created can be given as arguments based on the number of cores present in the machine. Scala helps us to clearly separate the number of cores available and number of threads that can be created. Even

though number of threads that can be created exceeds the cores, at a given time, number of threads that are executing will be equal to the number of cores given as input.

Getting a return value from scala Actors using Messages: When work is given to worker actors, we can get the return value from them using messages. This makes using scala actors more convenient compared with other languages like Java. For example, in Java, when a thread is given a run() method its return type is void. We cannot get any reply from that thread. There we can use some global variable to store the result computed by the thread. Using scala actor system, all this mess code can be removed and we can get return values nicely through messages.

Using futures to get a result from an actor: If the caller is an actor, it can send a message to the caller. Otherwise, scala provides the concept of **Future.** Using futures, we can specify the caller thread that a result will be given to it in future. Now we can block the caller thread if we need the result to proceed further, using Await.result(future, timeout) method provided by scala.

Conclusion:

We have implemented Connect4 game using scala. We induced parallelism using scala actor system and evaluated the performance. The results of the experiments are as expected. Scala provided a good abstraction of actors to visualize the whole process of parallelizing.

Future work:

We have implemented parallelism at first depth as explained in Performance Analysis section. Future work can be to induce parallelism at further depths depending upon number of cores. If more cores are available, parallelism can be induced for some more search space. For example, for a 7x7 matrix, at depth = 2,we will have 49 cases. If we have powerful machines having 20 cores, we need to evaluate some of the cases in parallel to make efficient use of the machine.

References:

[1] https://en.wikipedia.org/wiki/Connect_Four
[2] https://en.wikipedia.org/wiki/Minimax
[3] http://docs.scala-lang.org/overviews/parallel-collections/overview.html
[4] https://en.wikipedia.org/wiki/Scala_(programming_language)
[5] http://doc.akka.io/docs/akka/2.0.1/intro/getting-started-first-scala.html