A Distributed Chess Playing Software System Model Using Dynamic CPU Availability Prediction

Khondker Shajadul Hasan[1]

[1]School of Computer Science, University of Oklahoma, 110 W. Boyd St., Norman, OK 73019, USA
shajadul@ou.edu

Abstract - *Traditionally chess gaming software takes considerable amount of time in calculating the optimal move that can be made by a computer system. Most computer systems that make such calculations use some version of the minimax search with alpha beta pruning. Minimax with alpha beta pruning can be made distributed with principal variation splitting. The system can thus be used to reduce the time necessary to calculate the optimal move. Additionally, to select the distributed node containing less workload from the grid, a dynamic CPU availability prediction can be deployed to enhance the performance of task (searching game tree to find the optimal move) execution. An algorithm for dynamic monitoring of CPU has been proposed and discussed in this paper besides PV-Splitting minimum and maximum algorithms. Prediction of CPU availability is important in the context of making task assignment and scheduling decisions. Extensive experimental studies and statistical analysis are performed to evaluate the performance improvement of the model using the chess software system which uses the PV-Splitting for task parallelism.*

Keywords - Adversarial Search, Alpha Beta Pruning, Dynamic CPU Availability, Minimax Algorithm, Multi-core processor, Principal Variation Splitting.

1 Introduction

Chess has become very common game and nearly available on any kind of computer. As chess is in existence for very long time, much of that time, it has been played between human players. Only in recent times (relative to the time chess has been played) have computers started to play chess. The idea of a non human playing chess came to realization with the advent of digital computers. Even with the advent of digital computers, the computer could not play chess well. This was mainly due older computers not having the necessary computational power to play chess [1].

With the improvement in computational power and advanced algorithms, computers now have better chess playing software. However, when playing chess on personal computers, it takes a lot of time to calculate its move for advanced levels and with lack of computational power; it is not enough to challenge the best human players. So, computer scientists turned to distributed processing for improving this scenario [2]. Parallel processing allowed a massive increase in computational power which is needed to challenge good chess players. If the time needed for the computer to calculate a move could be cut, then the game against a computer could be more enjoyable. Our objective in this paper is to reduce the time need by a computer to calculate moves. In 1997, IBM's Deep Blue supercomputer created a sensation by defeating then world champion Gary Kasparov. This caused the world to take computer chess seriously. Nowadays, computer chess has become very common. Chess software is available on nearly any kind of computer.

To reduce the immense searching time for calculating moves, distributed computing has been employed. The processing power of two or more computers connected over a LAN was used to reduce time needed to calculate a move. The part of the program that performs the actual computations will run in the background of the various computers being used. This will allow the computers to be used for other, non cpu intensive purposes, while the computation is being done in the background.

It is necessary to divide the task of finding an optimal move for the computer of a grid to make chess distributed. To do this, Principal Variation Splitting, or PV-Splitting has been employed. PV-Splitting is an algorithm that was originally designed for shared memory multiprocessor systems. PV-Splitting had to be changed slightly in this paper to make it run in a distributed system.

2 The Distributed Model

In this paper, distributed computing have been used in which three or more computers connected over a LAN to reduce time needed to calculate a move with high processing power. The model of the distributed

chess system consists of two types of nodes: a single parent node and two or more worker nodes. The parent node divides the job of calculating the optimal move for the computer to make into smaller jobs. These jobs are then assigned to the worker nodes to compute. These jobs are assigned when messages are passed from the main node to the worker nodes using a local area network. The worker nodes are implemented as a Microsoft Windows service. This allows the worker node to run in the background.

The jobs are divided using principal variation splitting. Each job consists of searching the sub tree of a particular node as its root [4]. Different jobs do not have any intersection. This means the sub trees that are searched do not form a directed acyclic graph. Two different jobs with different root nodes do not have anything in common. Thus there is no communication necessary between workers searching different sub trees.

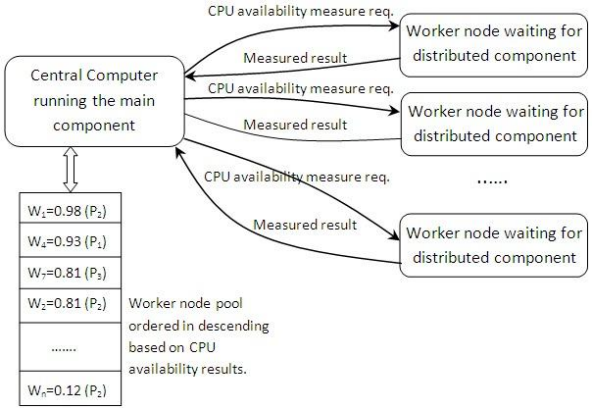

Figure 1: Distributed System Model.

Figure 1 contains the distributed model for the chess game system. The central computer which is running the main component of the game will dynamically measure the CPU availability of each worker node [6]. The measure data is used for arranging the worker nodes into the computer pool. The worker node which has the highest CPU availability gets the highest priority of achieving task as it can process early.

Each sub tree is searched using the minimax [1] algorithm with alpha-beta [2] pruning. When a job is assigned, the alpha beta value of the sub tree's root node's parent node is used. Once a job is finished, the min or max value of the sub tree's root node is returned as a message. The alpha beta values of the parent are then updated. Since there is only one main node in our distributed system, it will reside in one computer. The many worker nodes may reside in more than one computer. There may be a main node and one or more worker nodes on the same computer. There may also be more than one worker node per computer. This can be used for performance gains for computers with multiple CPUs or multiple cores.

Also, all the worker nodes can reside on a single computer. The system will not be distributed, but will be able to take advantages of multi-CPU or multi-core processors. It will also not suffer from any network lag.

3 Principal Variation Splitting

Principal Variation Splitting [3, 4] or PV Splitting was originally designed for multi-processor systems. In this case, it was adapted to work on a multi computer distributed system. PV Splitting is ideal for our model of the distributed system, as PV Splitting allows the job to be divided into smaller jobs from one main node and then have the job assigned to different workers.

The PV Splitting searches the same tree as the minimax algorithm [4]. It does the search in a parallel fashion, or in this case using multiple computers in a distributed system. PV Splitting is designed for the case when alpha beta pruning is used along with minimax search. In PV Splitting, starting with the child nodes of the root node, the leftmost child node's sub tree is searched first and then the remaining child nodes are then searched. This is done recursively. That is when searching the leftmost sub tree, the leftmost child node is expanded into its children first. Among its children, the leftmost child's sub tree is again searched first before searching the remaining children in parallel.

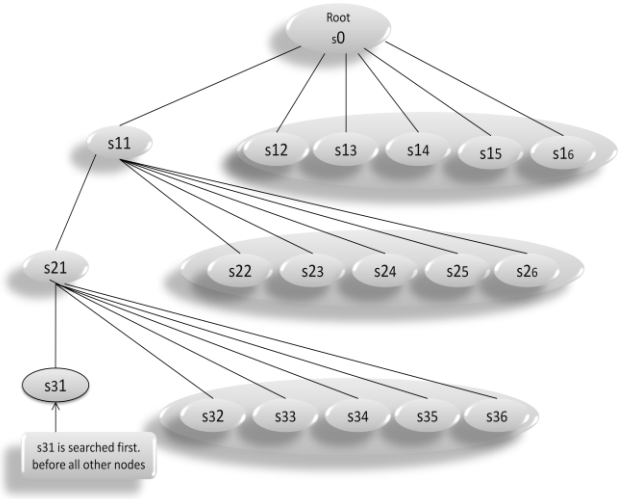

Figure 2: Principal Variation Splitting Procedure.

This is illustrated in Figure 2. The root node s0 is expanded into its children s11, s12, s13, s14, s15, s16. The left sub tree, s11 and all its descendants are searched first before the other children of s0 are

searched in parallel. The node s11 is expanded into s21, s22, s23, s24, s25, s26. Again, s21 and its descendants are searched before the remaining children are searched in parallel. The s21 node is then expanded into its children s31, s32, s33, s34, s35, s36. The node s31 is searched first before all other nodes.

Once the search of the sub tree of s31 has been finished, then sub trees of its sibling nodes s32, s33, s34, s35, s36 are searched in parallel. Similarly, once the search of s21 is complete the siblings of s21: s22, s23, s24, s25 and s26 are then searched in parallel and so on. Finally, when these searches are complete, their parent s0 has been searched completely. Thus the entire tree has been searched. All searches of a sub tree are done with minimax and alpha beta pruning.

Here is the proposed PV-Splitting [5] algorithm adapted for distributed systems:

```
Function PVSplit (start_state,)
Begin
    r := PVSplit_Max(start_state, -infinity, +infinity)
    return r
End
```

```
Function PVSplit_MAX(state, alpha, beta)
Begin
    IF (state is a leaf node)
        return evaluation_function(state)
        state[0]  = first child of state
        IF(current depth < max PV-Split Depth)
            r = PVSplit_MAX(state, alpha, beta)
        Else
             r = MAX(state, alpha, beta)
            alpha = r
            max = r
            state[1..N] = children of state, not including
                          state[0]
            For i = 1 to n in parallel
                (Wait for a worker)
                r = MIN(s[i], alpha, beta) /*MIN
                being minimax algorithms min executed on
                a worker */
                        IF r > max
                                max = r
                        IF r >= beta
                                prune tree
                        IF r > alpha
                                alpha = r
    return max
End
```

Figure 3: Algorithm of PV-Split Max modified for distributed system.

Next function, PVSplit_MIN algorithm for distributed system is given below:

```
Function PVSplit_MIN(state, alpha, beta)
Begin
    IF (state is a leaf node)
        return evaluation_function(state)
        state[0]  = first child of state
        IF(current depth < max PV-Split Depth)
         r = PVSplit_MAX(state, alpha, beta)
         ELSE
             r = MAX(state, alpha, beta)
             beta = r
             min = r
             state[1..N]  = children of state, not
               including state[0]
             For i = 1 to n in parallel
                (Wait for a worker)
        r = MAX(s[i], alpha, beta)  /*MAX
        being  minimax algorithms min executed on a
        lightly loaded worker */
                    IF r < min
                            min = r
                    IF r <= alpha
                            prune tree
                    IF  r < beta
                            beta = r
    return min
End
```

Figure 4: Algorithm of PV-Split Min modified for distributed system.

Each sibling in a parallel search [3] is searched using a worker. Each worker has the ability to search a sub tree using minimax and alpha beta pruning. When the parallel search is started, each sibling node attempts to acquire a free worker. To make sure that sibling nodes do not attempt acquire more workers than exists in the system, a semaphore is used. Once a node acquires a worker, it instructs the worker (through message passing) to search the sub tree of its node. Once the search is done, the worker returns the min/max value for the node as well as other search statistics using message passing. Using the min/max value of the sibling node, the min/max value of the parents is updated as well the alpha beta values.

One of the shortcomings of PV Splitting occurs when the number of worker nodes is greater than the number of sibling nodes that are to be searched. Some of the workers will remain idle if this were the case. The average branching factor for chess is over 30. So on average, for over 30 workers, some of the worker would remain idle.

4 The Chess System Model

A distributed chess game which can select worker nodes for processing its task (searching game tree for optimal move) is modeled using the CPU availability prediction. In this system, distributed computing have been used in which three or more computers connected over a LAN to reduce time needed to calculate a move with high processing power. The part of the program that performs the actual computations will run in the background of the various computers being used. This will allow the computers to be used for other, non CPU intensive processes.

A static model can be deployed to predict the CPU availability for distributed environment. Static prediction model depends on prior information about the number of CPU bound tasks *(n)* in the run queue, the number of I/O bound task *(m)* in the run-queue, and the unloaded CPU usage ($\hat{U}_i$) value for these last processes [7]. As I/O bound processes cannot use the processor all of this time, the CPU-bound processes steal them for some fraction of their quantums. On average, this behavior is coherent with the Linux scheduler mechanisms. The Linux scheduler provides higher priority to I/O bound process than CPU bound process by assigning I/O bound process in higher priority queue.

If the number of CPU core is *c*, the number of hardware thread is *HT*, and the number of run-queue is *R* then simply *R = (c*HT).* That is, for each logical thread (hyper-thread) there is a separate private run-queue. As the static model derived for measuring CPU availability is based on the prior information about the run-queue, for a dual-core processor with two hyper-threading requires four separate predictions (one for each logical processor). Similarly, for a multi-core system, for each logical processor we need to measure separate CPU availability predictions. Therefore, the definitive expressions for the static prediction model to compute the CPU assignment prediction for a new process, denoted by α_{lp} where $lp \in \{1, 2, ..., R\}$, is:

$$\alpha_{lp} = \begin{cases} \frac{1-\sum_{i=1}^{m} U_{i,lp}}{n+1} & if \ \frac{1-\sum_{i=1}^{m} U_{i,lp}}{n+1} > \frac{1}{n+m+1} \\ \frac{1}{n+m+1} & otherwise \end{cases} \quad (1)$$

with the expression of shared CPU usage is as follows:

$$U_{i,lp} = \frac{\hat{U}_{i,lp}}{1+\sum_{j=1}^{m} \hat{U}_{j,lp}} + \frac{\hat{U}_{i,lp}^2}{m+1} \cdot \sum_{j=1, j\neq i}^{m} \left(1 - \hat{U}_{j,lp}\right) \quad (2)$$

These equations will provide prediction of CPU availability for each logical processor (hyper-thread) of any *c-core* processors. Here, we can see that the first part of the equation (2) computes the shared CPU usage of the I/O bound process in the run queue including its own time and time stolen from other processes. The second part of the equation computes the usage of the CPU bound process in the run queue. As this model is derived by observing a number of scheduling pattern in different and by taking the average, it can be concluded that the correlation based empirical models has been used in time series analysis for the first situation.

Instead of assigning task to a random available worker node, this model relies on a dynamic prediction monitor which is a real time monitoring utility responsible for periodic measurements of states and available resources for the worker nodes. The available resource data of worker nodes measured by dynamic monitor[7] can be used to calculate the CPU availability of each node while placing it into the computer pool. Figure 5 contains the algorithm for measuring the CPU availability of worker node dynamically.

```
While (CheckMate ≠ true) do
 IF (WorkerNodes > SiblingNodes of GameTree) then
  For (each worketNode in the pool) do
   Obtain the PID's of the N processes in the run-queue
   For ( i=1 to N)do
    Access the node's environment variable to obtains: UserTime, SystemTime, StartTime,
    and ProcessState
    IF (This process has been assigned to the processor in the last T) then
     IF (The process is in the monitor queue) then
      Calculate the shared CPU usage considering: UserTime_old, SystemTime_old, StartTime_old
      IF (ProcessState ≠ running) then
       ProcessType=I/O-bound
      Else
       ProcessType=CPU-bound
     Else
      Calculate the shared CPU usage excluding: UserTime_old, SystemTime_old, StartTime_old
      IF (ProcessState ≠ running) then
       ProcessType=I/O-bound
      Else
       ProcessType=CPU-bound
    IF (ProcessType=I/O-bound) then
     I/O-Count = I/O-Count+1
    Else
     CPU-Count=CPU-Count+1
    Update the monitorQueue: UserTime_old, SystemTime_old, StartTime_old, and ProcessType_old
  WNPool ← Evaluate the dynamic CPU availability with I/O-Count, CPU-Count, and shared
           CPU usage for that process.
  Sort the worker node pool in descending order based on processor availability results.
  Wait until the next set of tasks are available in the queue.
End
```

Figure 5: Algorithm for CPU availability prediction in distributed environment.

When new tasks are available, they are assigned to worker nodes waiting in the computer pool. Computer pool contains all available workers which has finished their assigned task and waiting for a new task. The worker nodes in the pool are arranged in descending order based on CPU availability percentage. Suppose there are three tasks waiting to be searched in distributed fashion. There are four computers waiting in the computer pool with CPU availability of 97%, 93%,

89%, and 62%. The first worker node which has CPU availability of 97% gets the highest priority as it can process the assigned task faster. We can pick tasks randomly but the worker node is selected based on the CPU availability. If the number of task is more than the available worker nodes in the pool, then tasks can be assigned randomly to any worker node. Figure 6 shows a typical task assignment procedure in which three tasks are assigned to first three waiting nodes in the pool.

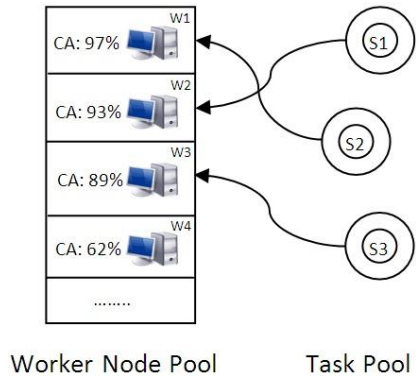

Figure 6: Task assignment to worker nodes based on CPU availability.

In games such as chess, there is a game tree consists of all possible moves by both players. Depending on the resulting board position, we can search this tree to find an effective playing strategy. Game playing is a multi-agent environment where each agent's search must try to put itself in a position to win while also preventing the opponent from putting it in a position where it is able to win. This kind of search between opposing agents is called adversarial search.

The chess game is deterministic, turn-taking, two-player zero-sum games of perfect information. So, agents have complete knowledge of the game state and the scores at the end of the game are opposite and sum to zero. As agents in games generally have a limited amount of time to make a move, so efficiency in finding a good move is important. Pruning the search tree to reduce search time, thus becomes very important.

Figure 7 shows the flow chart of the chess game of two players (between human and computer). After a human move, it always checks whether the move is legitimate move or not. If move is legal then it will calculate new move and change the board position. If it's a checkmate then the game is over else it will continue calculating new moves until one player gets checkmate and the game is over. The system will display the check mate and terminate.

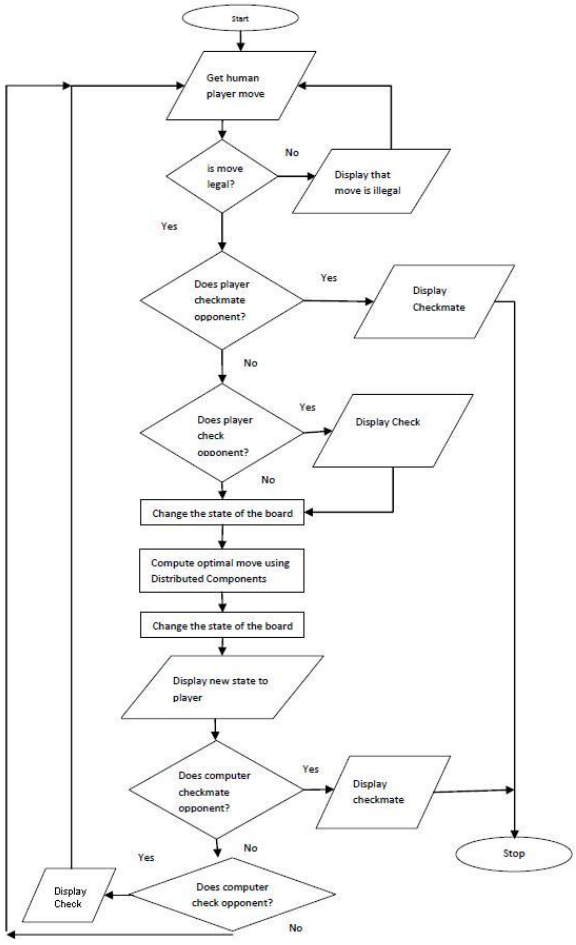

Figure 7: Program flow control depending on board position.

Figure 8 shows the GUI of the chess game board implemented during this project. This chess board can

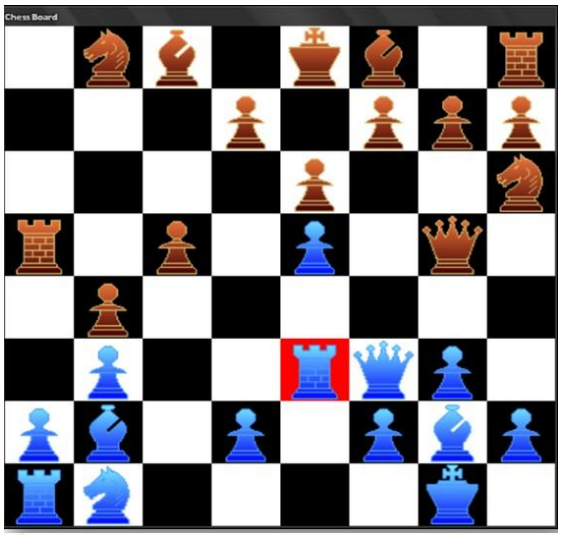

Figure 8: An instance of the implemented Chess game board position.

be operated using keyboard or mouse or both. After each legal move, the system calculates new move and updates the board pieces based on the move. It can also identify any illegal move provided by user and prompts accordingly.

The next section contains the results of several empirical studies for measuring the performance of the chess game implemented for this project in a single computer and distributed environment.

5 Analysis of Experiment Results

As the objective of the project was to create a distributed program that would reduce the time necessary to calculate moves, the system was evaluated for the amount of time taken by the program in two different scenarios. In the first scenario, one computer was used to calculate the moves. In the second scenario, three computers were used to calculate the same move.

First a set of states from which to perform the evaluations were created. Each state is a certain configuration of pieces on the board. For each of the states, move was initiated by the human player and then the time taken for the system to respond with its move in both scenarios was determined. For the first scenario a move was made and then the time taken system to make its move was observed. In the second scenario, the same move was made from the same state. The time required to make the move as well as the number of nodes searched by each computer was measured.

In the first scenario, a single pc was used and performed moves for certain states. These states are labeled 1, 9, 15, and 24. In the second scenario, the same states 1, 9, 15, and 24 were used again. In all states a remarkable decrease in the times required to calculate a move was seen. In state 1, the time decreased from 35.48 seconds to 12.5 seconds. In state 9, the time decreased from 45.07 seconds to 20.29 seconds. In state 15, time decreased from 69.87 seconds to 26 seconds. And in state 24, the time decreased from 40.06 seconds to 16.25 seconds. The results are shown as a bar graph in Figure 9.

As part of node searching, the average search speed on one PC in the first scenario was 421,946 nodes/sec. In the second scenario, involving three computers, the search speed was always over 1,200,000 nodes/sec. Thus the search speed was increased over three times by using three computers instead of one. It should be noted that the computer resources used were not homogenous. In the benefit of load balancing, the faster computers got more nodes to search. It has been observed that the dual core machine has searched the most nodes because it was the fastest.

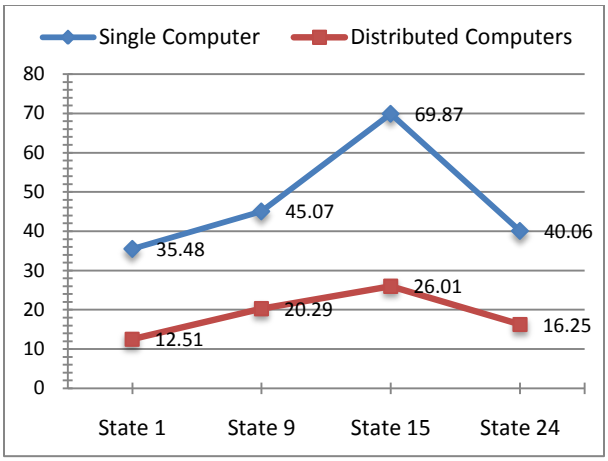

Figure 9: Comparison of time taken (in second) by the distributed system and a single computer.

The empirical results shows that the number of nodes searched by a single computer is always less than the number of nodes searched in distributed environment. This difference can reach as high as 18,708,420 nodes. Searching more nodes always helps to find better optimal solution. Therefore, this distributed system can provide much more competitive chess game than an individual computer. Figure 10 shows a comparison of the number of nodes searched between single computer and distributed system.

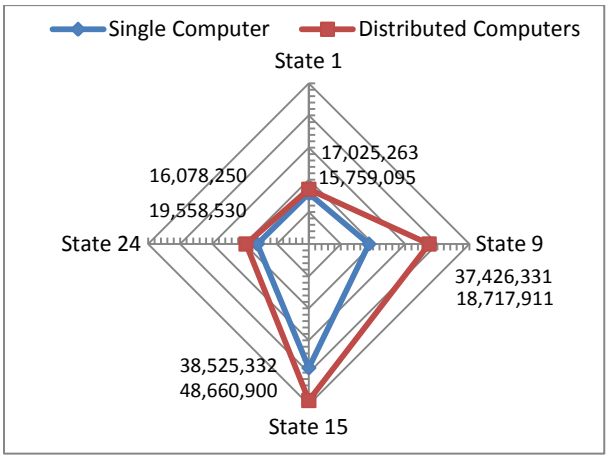

Figure 10: Comparison of total number of nodes searched by the distributed system and a single computer.

It should be noted as well, that the computer resources used were not homogenous. The different computers have different computational capabilities. PC#1 was an Athlon XP 2.4GHz, PC#2 was a Pentium 4 and PC#2 was a Core Duo 1.6GHz. In the interest of load balancing, the faster computers got more nodes to search. It can be seen from the table below that the dual

core PC#3 searched the most nodes because it was the fastest (summing nodes searched over the two cores).

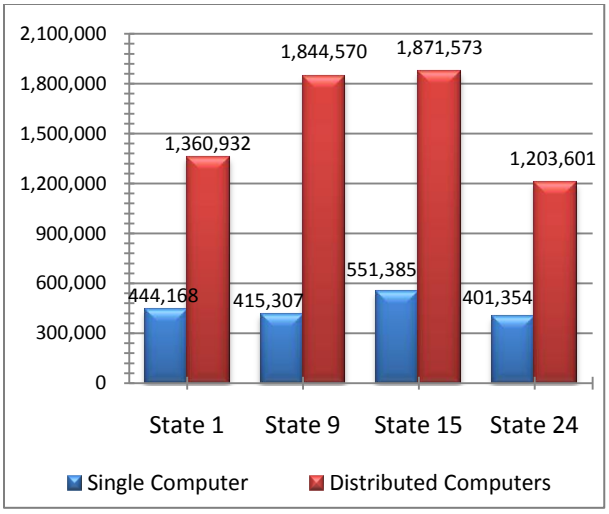

Figure 11: Comparison of search rate (nodes/ second) between a single computer and distributed system.

The rate at which nodes wear search was also calculated. The results are shown as a bar graph in Figure 11. The average search speed on one PC in the first scenario was 42,1946 nodes/sec. In the second scenario, involving three computers, the search speed was always over 1,200,000 nodes/sec. Thus the search speed was increased over three times by using three computers instead of one. (It should be noted that one of the computers was a dual core and both cores were used).

6 Conclusion

This paper has developed an analytical model (and conducted empirical studies) for predicting the CPU availability of worker nodes (single- and multi-core processors) for enhancing the performance of a distributed chess software system. This paper is a distributed computing effort in the field of artificial intelligence. The PV-Splitting algorithm modified for distributed systems was implemented and was validated. The results of the empirical study were quite impressive. The optimal moves made by the computer in response to human player moves takes far less time than while the game was running in a single computer.

The empirical studies conducted in this paper shows that the new system significantly reduces the optimal move calculation time when PV-Splitting has been employed along with dynamic CPU availability predictions. The system was tested in a case of one computer vs. three computers. The results show a large increase in speed, where speed was calculated as the number of nodes searched per second. Thus the amount of time taken to calculate a move was decreased, which was the main objective of the system.

As part of future work, we can focus on few deficiencies of the system like it cannot take advantage of more worker nodes than the number of branches of a node. The average branching factor is 35. This is a flaw of the PV-Splitting algorithm. The minimax implementation in the worker nodes can also be improved using Negascout algorithm (a directional search algorithm which is faster than alpha-beta pruning) to provide faster performance.

6 Reference

[1] Burkhard Monien, Thomas Ottmann, Data Structure and Efficient Algorithms, Springer Press, 1992.

[2] Stuart Russell, Peter Norvig, Artificial Intelligence: A Modern Approach, Amazon Press, 2008.

[3] Valavan Manohararajah, "Parallel alpha-beta search on shared memory multiprocessors", Master's thesis, Graduate Department, of Electrical and Computer Engineering University of Toronto, Canada, 2001. http://www.valavan.net/mthesis.pdf

[4] Khondker S. Hasan, Alok Chowdhury, Asif Mahbub, Ahammed Hossain, Abul L. Haque, "Implementation of a Distributed Chess Playing Software System Using Principal Variation Splitting", *16th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, July 2010, Nevada, USA.

[5] Yaoqing Gao, T. A. Marsland, "Multithreaded Pruned Tree Search in Distributed System" http://www.cs.ualbetra.ca/~tony/Recent Papers/icci.pdf

[6] Dingzhu Du, Panos M. Pardalos, Minmax and Applications, Amazon Press, 1995.

[7] Martha Beltrán, Antonio Guzmán and Jose Luis Bosque, "A new CPU Availability Prediction Model for Time-Shared Systems", *IEEE Transactions on Computers*, Vol 57, No. 7, pp. 865-875, ISSN: 0018-9340, July 2008.

[8] R. Wolski, N. Spring, and J. Hayes, "Predicting the CPU Availability of Time-Shared Unix Systems on the Computational Grid," *Proc. Eighth International Symposium on High Performance Distributed Computing*, pp. 105-112, August 2002.

[9] *Operating System Concepts*, 8th Edition, Silberschatz, Galvin, Gagne, Wiley & Sons Inc., ISBN: 978-470-12872-5, July 2009.