$\alpha\beta$Service - An Asynchronous Parallel Tree Search Service for ChessGrid.

by

Sukanta Pramanik

B.Sc.(Engg), Bangladesh University of Engineering and Technology, Dhaka

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

July 2005

Abstract

The performance of game playing programs depends heavily on the strength of the search algorithm used. As such, several research activities have been conducted to speed up game tree search algorithms by using parallel machines. Most of these algorithms are targeted towards shared memory models or tightly coupled systems. In recent years, the rapid progress of Grid computing has led to finding a Grid solution to difficult computational problems. To ensure interoperability among diverse computational resources in the Grid, Service Oriented Architecture is used to achieve loose coupling among interacting software agents, known as *services*. Finding an efficient tree search algorithm in this new paradigm is a challenging problem.

This thesis proposes a new service oriented algorithm, called $\alpha\beta$Service, which targets loosely coupled systems. It creates a tree of services to search the game tree concurrently. The necessity of synchronization points is eliminated with a notification mechanism, enabling the concurrently running services to continue their asynchronous search without waiting for others to finish. The $\alpha\beta$Service is implemented as a Grid service in Globus using Globus Toolkit 3. The service is deployed to create a ChessGrid testbed and experiments are conducted with a branching factor controlled test suite which demonstrates an average speedup of 2.61 with six machines.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my gratitude to Son T. Vuong for proposing this research topic. He has been my advisor, supervisor as well as a personal friend and his cooperation, interest and constructive criticism have been invaluable throughout the course of this thesis.

A big thanks to Charles "Buck" Krasic and Alan Wagner for allowing me to use the Netbed cluster. Buck has been of tremendous help during the setup of the ChessGrid in the cluster. Thank you also for being the second reader and providing some valuable feedback on the initial version.

I would also like to thank the other members of the UBC NICLab and DSG group for many helpful discussions. My special thanks to Brendan Cully for all the helps in Linux tweakings, Christian Chita for some useful insights and Ying Su for letting me run a server in her desktop.

SUKANTA PRAMANIK

The University of British Columbia
July 2005

Chapter 1

Introduction

Chess has fascinated people throughout the world for centuries. Though it dates back to antiquity, there is debate as to which culture its origins should be credited. The most commonly held belief is that Chess originated in India, having spawned from the game *Chaturanga* (from Sanskrit *Chaturangam*, meaning 'four arms' or 'four members')which appears to have been invented in the 6th century A.D. The modern era of chess, however, may be said to date back to about the 15th century, when the pieces gained their present form and standardization started. [TCE01, WIK05]

Ever since the old days of punch-card computers, people have also been fascinated with chess-playing programs. In the public eye advances in chess-playing computer programs have become analogous to progress in AI. The first article on programming computer for playing chess is due to Claude E. Shannon [Sha50]. In 1950 Shannon noted the theoretical existence of a perfect solution to chess and described two general strategies.

> Type A - expand all sequences of possible moves out to a fixed number of levels and combine the evaluations of these sequences in a simple tree computation thereby using the combined evaluation to choose the best move.

Type B - only perform selective expansion of certain lines, using knowledge to prune uninteresting branches.

Shannon advocated that Type B is closer to the way humans play chess. Turing's 1951 program [Tur53] was a Shannon Type B program that only expanded moves involving captures. For the 1973 Computer Chess Championships Slate and Atkin [SA77] used a Type A search routine, Chess 4.0, that won comfortably and the switching to Type A started. Debate to which one is better still continues, although most modern leading programs use a hybrid of the two types. In 1996 Deep Blue became the first computer system to win a chess game against a reigning world champion (Garry Kasparov) under regular time controls. An upgraded Deep Blue later became the first computer system to defeat a reigning world champion in a 5-game match in 1997.

1.1 Motivation

Playing strength of a Chess program is highly dependent on the depth upto which the game tree is searched. Due to this continuous necessity of stronger program and hence deeper lookahead, multiple processors were employed to speedup the search. Interestingly, almost all of these attempts use a synchronous algorithm to parallely search the tree.

The sequential algorithms use a pruning technique to minimize the number of nodes searched. After it completes searching one subtree, a bound is placed on the possible return value of subsequent subtrees which is then used to prune redundant nodes. This concept was also incorporated in the parallel approaches as the processors were forced to complete searching one part of the tree before continuing to the next. This, in effect, keeps the number of nodes searched by the parallel algorithm close to the sequential implementation and perhaps is the main reason why asynchronous approaches did not gain popularity.

However, this also brings in the global *synchronization point*s along the search path to which all the concurrently running processors must reach before they continue any further. The advantages of synchronous algorithms, i.e. searching nearly the same number of nodes as its sequential counterpart, is seriously undemined by the processors sitting idle at the synchronization points. The situation is further aggravated when we have more processors than divisible works to do.

Synchronous algorithms also use a distributed shared table to ensure that searching similar nodes is not repeated on multiple processors. Most of these algorithms are tested on a tightly coupled system and in a loosely coupled environment or in the absence of this table they do not show the reported parallel efficiency.

Newborn first suggested an asynchronous solution to the problem as the concurrently running processors searched the partition of the tree delegated to them independently. When the time limit exceeded all the results were combined to get the best move from the root. Newborn's UIDPABS [New88] has limited scalability as the tree partition occurs only at the root. This idea of partitioning is generalized in Brockington's APHID [BS00] that uses a master process to distribute the nodes at a certain depth among child processors which then carry out the asynchronous search. Aside from these two works, no other research has focused on asynchronously searching a distributed game tree.

Almost all the synchronous algorithms are targeted towards very tightly coupled clusters of computers. In recent years Grid computing has evolved into a major discipline in itself by its increased focus on coordinated resource sharing and problem solving in multi-institutional virtual organizations. [FKT01] A major shift in Grid is also happening towards service orientation, open standards integration, collaboration, and virtualizaiton to achieve loose coupling among interacting software agents in this heterogenous environment. Globus Toolkit 3 brought in the Open Grid Services Architecture (OGSA) [FKNT02] and thereby aligning itself with the new emerging Service Oriented Architecture. While the Grid concept is trying to

achieve a computing paradigm similar to an electric power grid with a variety of heterogenous resources to share, finding an efficient tree searching algorithm that can exploit the functionality of this environment is a challenging problem. To the author's knowledge no attempt has been made yet to develop a service oriented parallel game tree searching algorithm. The goal of this thesis is to find a solution to this problem and compare the observed speedup with already existing solutions in other domains.

1.2 Thesis Outline

Chapter 2 and 3 are intended to be a brief overview of the related fields. Chapter 2 discusses existing game tree searching techniques in both sequential and parallel domain, focusing mainly on minimax based approaches. In Chapter 3 the concept of Grid is introduced and the functionalities of Globus Toolkit 3, the de-facto standard for building grids, are detailed. The chapter then continues to build the groundwork for our proposed algorithm by illustrating the ChessGrid's architecture and how the functionalities of Grid are exploited in the algorithm.

The algorithm for $\alpha\beta$Service is then presented in detail in Chapter 4. Chapter 5 shows the experimental results that we received by deploying the service in the ChessGrid testbed of six machines. A brief explanation of the terminology used to summarize the performance of a parallel program is also presented here. Finally Chapter 6 draws the conclusion and addresses a few possible future course of work.

Chapter 2

Game Tree Searching

2.1 Introduction

Games hold an inexplicable fascination for many people and the notion that computers might play and compete with men existed as long as computers. For *artificial intelligence* researchers, the abstract nature of games make them an appealing subject for studying, which is evident from the fact that it was one of the first tasks undertaken in this field.

The mathematical theory of games was invented by John von Neumann and Oskar Morgenstern [vNM44] as early as 1944. Game tree searching has also come a long way since the introduction of the Minimax algorithm followed by the $\alpha\beta$ pruning. However, the basic idea established by the minimax, and subsequently $\alpha\beta$ pruning, has been the cornerstone for most sequential as well as parallel algorithms.

This chapter briefly introduces the preliminary ideas in this field. Section 2.2 touches on some basic concepts of game theory. Section 2.3 introduces the concepts of Minimax, Negmax (a variant of Minimax) and $\alpha\beta$ pruning. Section 2.4 briefly introduces the existing parallel tree searching techniques, the main emphasis being on the minimax-based algorithms.

2.2 Basic Elements of Game Theory

A game consists of a set of rules governing a competitive situation in which from two to n individuals or groups of individuals choose strategies designed to maximize their own winnings or to minimize their opponent's winnings; the rules specify the possible actions for each player, the amount of information received by each as the play progresses, and the amounts won or lost in various situations. [vNM44]

The game that we are particularly interested in, i.e. chess, is known as *two-player zero-sum game with perfect information.* A *zero-sum game* indicates that if we add up the wins and losses in a game, treating losses as negatives and wins as positives, we find that the sum is zero for each set of strategies chosen. In less formal terms, a zero-sum game is a game in which one player can only be made better off by making the other player worse off in an equivalent amount. Tic-tac-toe is a simple example of such a game: any move that brings one player closer to winning brings the other player closer to losing, and vice-versa.

Another crucial aspect involves the information that the players have when they are playing. A game with *perfect information* requires that at every point where each player's strategy tells her to take an action, she knows everything that has happened in the game up to that point. This is the case for games like chess, othello, checkers, et cetera. On the other hand bridge and poker are examples of games of *imperfect information*, since some cards are hidden from the player and she must formulate her strategy in ignorance of this information.

The activity of game playing, in order to be a game, necessarily requires an opponent. In game theory, a two-player game can be formally defined by the following components [RN03]:

- The *initial state*, which includes the starting board position and the player to move first.
- The *position set*, which is the set of valid board states.

- A *successor function*, which defines the rules for moving from one state to another and given a board state returns a list of (*move*, *state*) pairs, each indicating a legal move to a resulting state.

- A *terminal test*, which identifies the end-game scenario, i.e. a state (*terminal state*) from where successor function returns a null set.

- A *utility function*, which gives a numerical value to the terminal state signifying the outcome of the game.

The initial state and the legal moves for each player define the *game tree* for the game. The *root* of the tree represents the current state(the initial state at the beginning) of the game. The *nodes* of the game tree will correspond to different positions and the *edges* will correspond to the moves from one position to a successor position. Each node can have any number of *children*, determined by the successor function, and this continues until we reach a *leaf*, which represents a terminal state. We assign a utility value to each terminal state based on the utility function(also known as objective function or payoff function).

In Figure 2.1 an example of a game tree for a simple game of tic-tac-toe is given near the end of the game. The two players are X and O and they take alternating turns at different levels of the tree. To each leaf in the game tree the utility function assigns a payoff value. We assigned a very simple payoff value- $+1$, 0 and -1 for a win, draw or loss with respect to the player X. The circled numbers near the internal nodes will be explained in the next section.

2.3 Sequential Game Tree Searching

2.3.1 Minimax and Negmax

The two players are named Max and Min based on their playing characteristics. Historically Max moves from the initial position and from there the two players

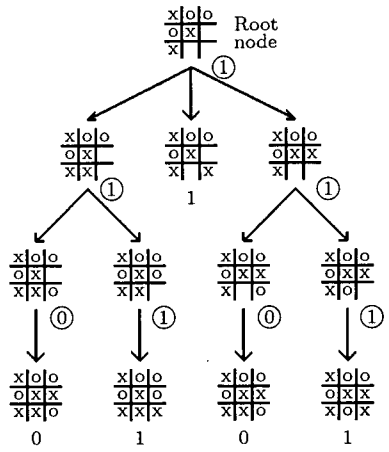


Figure 2.1: A simple game tree for Tic-Tac-Toe near the end of the game.

take alternating turns at different levels of the game. So nodes at even depth in the tree are called Max-nodes and nodes at odd depth are called Min-nodes. The common term for this is *ply*. The root node is said to be at ply 0, the immediate successors of the root node are said to be at ply 1, and so on.

At each Max-node, Max would like to play the move that maximizes the payoff, thereby choosing the maximum score among her children. On the other hand at each Min-node, Min will try to minimize the payoff, since that will maximize Min's payoff, and so she will choose the minimum of her children. In this way all the tree nodes can be assigned a payoff value or *minimax value*(the circled numbers near the internal nodes in Figure 2.1) starting from the leaves and moving bottom-up towards the root.

If p is a position from which there are n legal moves $p_1, p_2, \ldots, p_n$ and $F(p)$ is the minimax value attached to the node p, the search problem can be characterized as to choose the greatest possible value of $F(p)$. Based on our previous discussion

we can write this as,

$$F(p) = \begin{cases} f(p) & \text{if } n = 0, \\ max(F(p_1), F(p_2), \ldots, F(p_n)) & \text{if } p \text{ is a Max-node,} \\ min(F(p_1), F(p_2), \ldots, F(p_n)) & \text{if } p \text{ is a Min-node} \end{cases} \tag{2.1}$$

where $f(p)$ is the utility function.

It is clear from the above description that the minimax algorithm generates the entire game search space. However, when a computer is playing a complex game, it is impractical to search all the way down to the terminal states as the moves must be made in a reasonable amount of time. Shannon [Sha50] proposed that the search algorithm should be modified so that sufficiently deep positions are treated as terminal nodes. So the programs should *cutoff* the search at an earlier depth, often determined by available resources of time and memory. This strategy is called *n-ply look ahead*, where n is the number of levels explored, and the leaves of the generated sub-tree are called *cutoff nodes*. As the cutoff nodes are not terminal states, the utility function cannot possibly assign them a payoff value. Instead a heuristic *evaluation function* is applied to each leaf node, which gives an estimation of the position's utility. Thus the terminal test is replaced by a *cutoff test* and the utility function is replaced by the evaluation function.

An evaluation function estimates the expected utility of the game from a given position. Since we are not necessarily searching upto the terminal states, the performance of a game-playing program is dependent on the quality of the evaluation function. The value returned by the evaluation function should be strongly correlated with the actual chances of winning. The evaluation function should also be able to order the terminal states in the same way as a true utility function; otherwise, the program may select suboptimal move even if it computes the total search space.

Most evaluation functions work by calculating various features of the state, such as piece advantage, piece location, piece mobility, control of the center board,

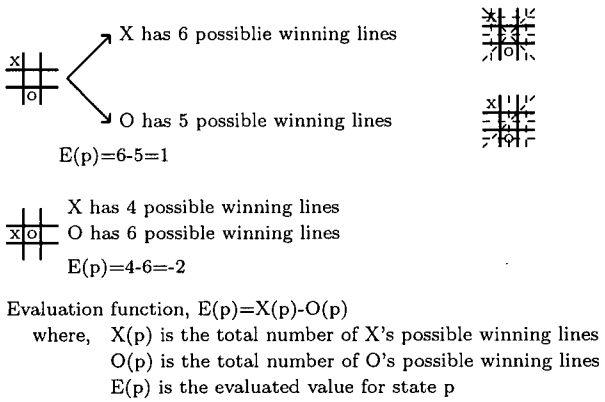


Figure 2.2: Computation of the evaluation function.[Nil80]

etc. They compute separate numerical contributions of each feature and then evaluate board states with a weighted linear function.

$$Eval(p) = w_1 f_1(p) + w_2 f_2(p) + \ldots + w_n f_n(p) = \sum_{i=1}^{n} w_i f_i(p) \tag{2.2}$$

where each $f_i(p)$ is a feature of the position p and w_i is specially tuned weights that tries to model the importance of that particular feature on the overall board evaluation. In a very simple evaluation function for chess, f_i can be the number of each kind of piece on the board and the w_i can be the values of the pieces(1 for pawn, 3 for bishop, etc.).

We conclude this discussion of evaluation function with another example of a game tree for tic-tac-toe, adapted from [Nil80], considering X as Max and O as Min. Here an evaluation function is used, which takes a state that is to be measured, counts all lines Max can make to win, and then subtracts from it the total number of winning lines for Min. This is illustrated in Figure 2.2. If a state is a forced win for Max, it is evaluated as $+\infty$; a forced win for Min on the other hand, as $-\infty$. The generated game tree is shown in Figure 2.3. To save space, cutoff is made at merely two-ply depth.

A little intuition shows that our minimax algorithm would require two different

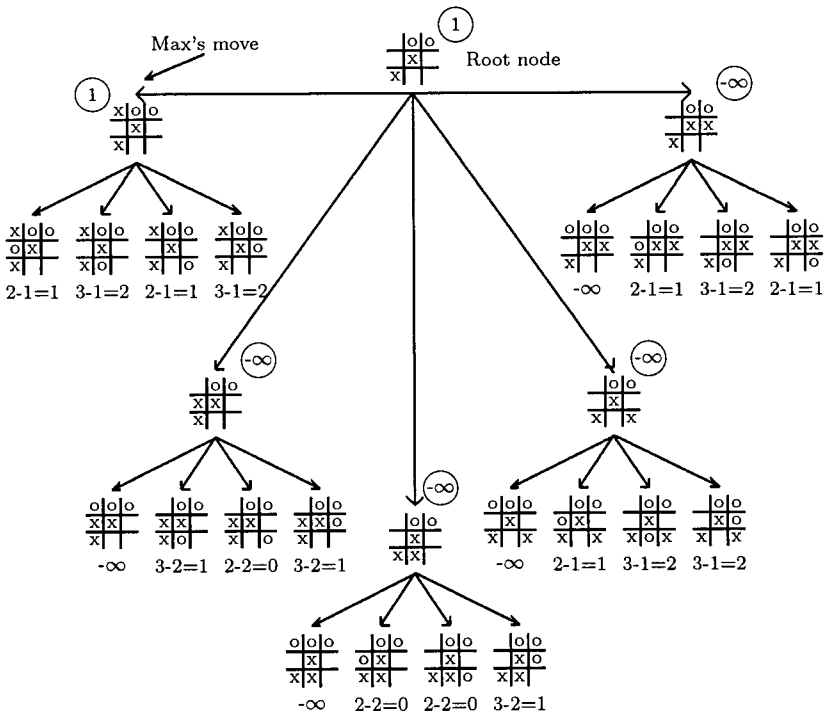


Figure 2.3: Two-ply minimax applied to X's move near the end of the game.[Nil80]

functions, $max()$ and $min()$, based on which player is making her move. However, with a little modification, we can suffice with only one. If p is a cutoff node with Max to move, evaluation function $f(p)$ represents its value. But if Min makes her move from p, its value is assumed to be -$f(p)$. With this formulation we can write our *negmax* equation as,

$$F(p) = \begin{cases} f(p) & p \text{ is a cutoff node} \\ max(-F(p_1), -F(p_2), \ldots, -F(p_n)) & p \text{ is an internal node} \end{cases} \tag{2.3}$$

2.3.2 $\alpha\beta$ Pruning

Both minimax and negmax pursue all the branches in space, including those that could be ignored or pruned by a more intelligent algorithm. It is possible to improve this brute-force approach by ignoring the moves which are incapable of being better than the moves that are already known. For example in negmax, if $F(p_1) = -a$, then $F(p) \geq a$. Now if p_2 has a legal move p_{21} such that $F(p_{21}) \leq a$, then obviously $F(p_2) \geq -a$. In that case we can prune the other moves from p_2 as we no longer need to know the exact value of $F(p_2)$. In any case, $-F(p_2) \leq a$ is always true and so it cannot improve the value of $F(p)$ further.

This line of reasoning brought in a method for game tree pruning which first appeared in late 1950s[NSS58] and paved the way for the $\alpha\beta$ *pruning*, the first account of which is due to Brudno[Bru63]. However, there are numerous claims about the first development of this algorithm; an excellent summary of which can be found in [KM75].

The basic idea of $\alpha\beta$ pruning is quite simple: there are some positions that the players will never reach as one of them will turn off from the path by choosing a move better for herself. To keep track of this, the $\alpha\beta$ search proceeds in a depth-first manner, with two bounds used at each node of the game tree, namely α and β. These two values, often referred to as *search window* and usually written as (α, β), are essentially the floor and ceiling, respectively, of the range of values the

search node might have. The α value, associated with Max nodes, represents the best choice (highest value) Max can guarantee herself by making some move at the current node or at some node earlier on the path to this node, given that Min will also do her best. Thus the value of α is monotonously increasing since it can be increased if we find a better path as we descend down the search tree. The β value, on the other hand, is associated with Min nodes and it represents the best choice (lowest value) Min can guarantee herself by making some move at the current node or at some node earlier on the path to this node. So, the β value is monotonously decreasing as we evaluate more branches under the Min node.

Algorithm 1 $\alpha\beta$ pruning pseudocode

```
function αβ(Node v, int α, int β): returns int
    if isCutoffDepth(v) then                    ▷ maximum depth is reached
        return evalFunction(v);                 ▷ evaluate and return
    end if
    Generate all the successors v_1, v_2, ..., v_n of v;
    if n = 0 then
        return evalFunction(v);                 ▷ evaluate the leaf node
    end if
    γ ← α;
    for i ← 1, n do
        score ← −αβ(v_i, −β, −γ);
        γ ← Max(γ, score) ;
        if γ > β then
            return γ;                           ▷ cutoff due to pruning
        end if
    end for
    return γ;
end function
```

This situation is shown in Figure 2.4. The picture shows how the utility value of a node is related to its reachability from the root. Since each player is trying to minimize her opponent's chances, the best value nodes of one player are never reached as the other player turns off the path to get to a lower valued node for the opponent. Max is constantly trying to increase the α value by finding a better move, while Min is trying to decrease the β value. If a node's value is between α

and β, then it is reachable from the root.

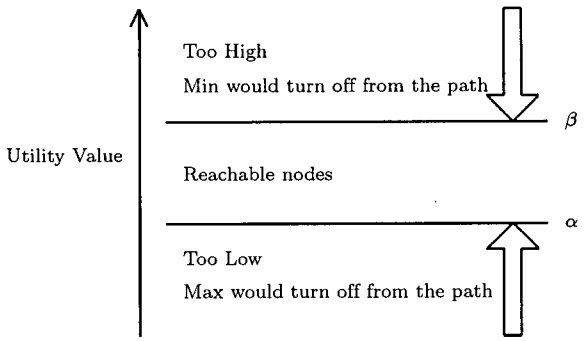


Figure 2.4: The influence of α and β

Since at the beginning we don't have any prescience about the node values, $(-\infty, \infty)$ is used as the initial search window. As we move down the tree each node starts with a search window passed down from its parent. At each node the α and β values are updated as we iterate over its children. In a Max node, α is increased if a child value is greater than the current α value and similarly at a Min node, β may be decreased. If we reach a point where $\alpha > \beta$, then we know that one of the players will never let the game reach this node and so we can prune the remaining children.

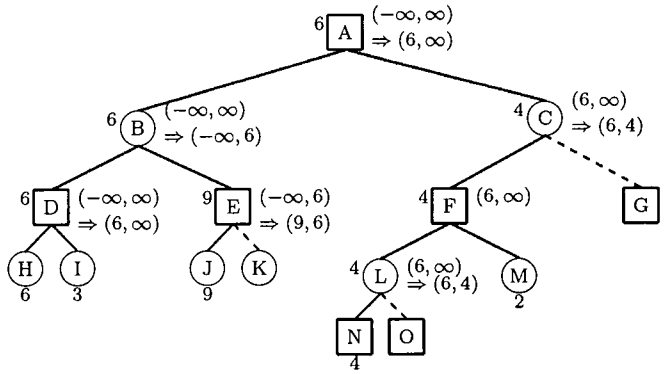


Figure 2.5: Shallow and deep cutoff

An example is given in Figure 2.5. The window with which the search routine first visits a node is given first beside each node and if updated in the later visit, the new search window is shown in the next line. As the depth-first search proceeds, the β value of B will be set to 6 after D is traversed. Being a Min node, B then has a value of *at most* 6. The next child of B, E, will then be called with a search window $(-\infty, 6)$. Now the first leaf below E has a value of 9. Hence, E, which is a Max node, has a value of *at least* 9. From our knowledge of B we know that Min will never choose node E as she already has a better move in D. Therefore, there is no point in looking at the other successors of E and we can prune them right away. Same thing happens at node C except that we are pruning Min's unreachable moves there.

The cutoff made at E and C are called *shallow cutoff* to distinguish them from another type of cutoff that can be obtained at even deeper levels and so is named *deep cutoff*. If we look at Figure 2.5 again, node C is called with a window $(6, \infty)$ and as we go deeper, at node L we come up with its first leaf having a value of 4. As L is a Min node, it then cannot have a value more than 4. Thus we can prune all the other successors of L as the α value set at A tells us Max has a better move somewhere else.

2.3.3 Enhancements to $\alpha\beta$

Numerous enhancements have been made to the basic $\alpha\beta$ algorithm since it was first evolved. [SP96] Among them those that are used or mentioned in the course of this thesis are briefly reviewed in the following section.

Iterative Deepening

Iterative deepening means repeatedly calling a fixed depth-first routine and search all the nodes that are at distance less than or equal to the fixed depth. With each repetition, the depth is increased until a time limit is exceeded or maximum search

depth has been reached. The idea with respect to the $\alpha\beta$ algorithm is that a k-ply search is completed before a $(k+1)$-ply search. Such a sequence of searches can be expressed as $S_1, S_2, \ldots, S_k, \ldots, S_N$, where S_k is a depth-first $\alpha\beta$-search of all move continuations upto the depth of k-plies.

The iterative deepening first started being used in chess programs by mid-1970's. [SA77] By using iterative deepening, search time for S_{k+1} can be estimated from the search time of S_k, although it may be far off the accurate value. On an average S_{k+1} takes about six times as long as S_k and this multiplier is generally proportional to the square root of the average branching factor. Now, as the branching factor can change abruptly based on some particular moves, so does the multiplier. A big advantage of iterative deepening is that, in a time constrained scenario, if S_k is not yet complete it can use the best move from S_{k-1}. A simple depth-search in this case can result in a disastrous move.

Aspiration Search

An $\alpha\beta$-search is normally called at the top level with a search window $(-\infty, \infty)$. Narrowing of this window can result cutoffs at earlier point of the tree, thereby evaluating fewer bottom positions compared to the $\alpha\beta$-search with infinite search-window. To narrow the search window, the value at the root of the tree is estimated by a lower-depth $\alpha\beta$-search. Thus if the value at the root is approximated to be v_{est} then an initial search window $(v_{est} - \epsilon, v_{est} + \epsilon)$ is used.

However, the improvement only happens when the returned minimax value, v_{real}, is contained in the narrow window that we use, i.e. $v_{est} - \epsilon < v_{real} < v_{est} + \epsilon$. If the minimax value is not in the narrower window used, i.e. $v_{real} \leq v_{est} - \epsilon$ or $v_{real} \geq v_{est} + \epsilon$, the whole game tree has to be researched with the search window $(-\infty, v_{real})$ or (v_{real}, ∞) respectively.

Move Ordering

In $\alpha\beta$-search, the search efficiency depends heavily on the order in which the moves are searched. It is essential that the best moves in a position are searched first so that the generated search window improves the number of cutoffs made by the algorithm. However, the dilemma lies in the fact that to know which moves are best we have to traverse them first. Usually some heuristic is used to order the moves from a position. One useful heuristic is to perform a shallow search, which obviously takes negligible time, and order the moves based on it.

Transposition Table

Often the same position can occur after several differing move sequences. If during the search, a value is assigned to one such position, storing it will become useful when at a later stage in the search the same position occurs again. The transposition table is a hashed repository of past search results and so can be used to detect identical positions in different branches of the tree. If a search arrives at a position that has been searched before and if the value obtained can be used (e.g. stored value is not from a smaller depth search), we can avoid re-searching. If the value cannot be used, it is still possible to use the best move that was used previously at that position to improve the move ordering. Transposition tables can speed up the search dramatically, particularly near the endgame.

Killer Heuristic

The killer heuristic is based on the fact that a good move in one branch of the tree is also good for another branch at the same level of the tree. These moves are known as killer moves and they are used to improve the move ordering. For this purpose at each ply a list of a few killer moves are maintained that are searched before the other moves. A bonus scoring system is stored sometimes for keeping the killer list sorted. Besides, a successful cutoff by a non-killer move can also overwrite one of the

killer moves for that ply. Slate and Atkin [SA77] noted that killer heuristic become more useful when a transposition table is being used because trying the same move early in the search at each depth will result in more repeated positions and thus more successful retrievals from the transposition table.

Null Window

When move ordering technique is used, it is often likely that the move tried first would contain the best move. This observation led to using a smaller search window for the remaining moves so as to reduce the size of the tree searched. Once the first move from a position in the tree returns a score γ, instead of searching the other moves with the window (γ, β), a null window (also called *minimal window*) $(\gamma, \gamma+1)$ is used. If a move leads to an inferior variation, it can be quickly shown if a move leads to an inferior variation. If, however, the value returned falls within the (γ, β) window then the sub-tree must be searched with the window (γ, β) to determine the correct value.

2.4 Parallel Game Tree Searching

It is well known that the efficient parallelization of $\alpha\beta$ is a difficult problem. [FF82] The difficulty in parallelization of $\alpha\beta$ stems from the fact that the algorithm is inherently sequential. After a subtree is completely searched, the payoff value obtained determines the search window for the subsequent subtrees which is then used for pruning. A naive parallel algorithm running a tree-node in a processor may end up searching the total sub-tree under it to later find out a better path in previous nodes. A sequential implementation would have never visited all these children due to a cut-off given by the previously computed bound on the score. This leads to the issues of the types of nodes and their relationship with the possible parallelism.

2.4.1 Analysis of the Tree Structure

The parallelism in $\alpha\beta$ is attained from searching different parts of the game tree at the same time. However, there is an inherent sequential nature of it as pruning depends on the complete searching of one subtree so as to be able to establish some bounds for the next subtrees. One has to flout this sequential model to obtain satisfactory parallelism. However, if it is totally overlooked then the algorithm may waste too much time on unpromising nodes thereby not searching upto a satisfactory depth.

Pruning in $\alpha\beta$ is based on the fact that we need not explore all of an opponent's response to our bad moves. As a matter of fact, if we can find only one refutation then we can prune the rests assuming that the opponent will also play optimally. This idea leads to the *perfectly ordered game tree* searching. Perfect move ordering means at any position the first move searched is the best move, or at least the refutation.

The complete analysis of a perfectly ordered game tree is due to Knuth and Moore [KM75]. In a perfectly ordered game tree the nodes can be subdivided into three following types as shown in Figure 2.6:

- Type 1: these nodes include all the best moves at a given position. The root is a type 1 node and so is the first successor of a type 1 node. Together they form the *principal variation*– the hypothetical line along which both player play their respective best moves.

- Type 2: all the child nodes of a type 1 node but the first are of type 2. Type 3 nodes, on the other hand, has all type 2 children.

- Type 3: all children of type 2 nodes are of type 3.

This classification is very important for parallel game tree search. At all type 1 nodes, the first child, which is also of type 1, has to be searched sequentially to set the (α, β) window. The rest of the children, which are all type 2 nodes, can be

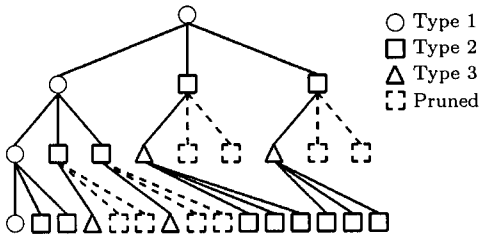


Figure 2.6: A perfectly ordered game tree.

searched in parallel. At the type 2 nodes, however, parallelism is unnecessary since the refutation would render all the other children unpromising. However, type 3 nodes can be searched fully parallely. This clear distinction breaks down for non-optimal trees, but it is still approximately correct and recent Chess programs have made much improvement on move ordering.

2.4.2 Parallel Search Techniques

From Baudet [Bau78] to Brockington [Bro98] a substantial amount of literature can be found that propose several different ways to parallelize chess playing programs, a nice taxonomy of which can be found in [Bro96]. However, the underlying strategies used in them to extract parallelism are often similar and can be categorized into three broad areas:

1. Parallel aspiration search [Bau78]: the initial $\alpha\beta$ window is partitioned into a number of contiguous disjoint windows, which are then used by different processors to search the total game tree.

2. Parallel node calculation: Hitech [Ebe86] used a 64-chip(one chip for each square) move generator, while Deep Thought [hH90], and subsequently Deep Blue [CJhH02] uses a 8x8 combinational array move generator with parallel hardware for evaluation.

3. Tree splitting [FF82]: the game tree is decomposed as its nodes are assigned

to different search processors.

The last method *tree splitting*, first introduced by Finkel and Fishburn [FF82], perhaps, is the most widely studied and used software approach to the parallelization and is the focus of this thesis.

2.4.3 Tree Splitting Based Algorithms

Tree Splitting

Finkel and Fishburn's *tree splitting* [FF82] has a static tree of processors which handles the top part of the game tree. The root in the processor tree evaluates the root position of the game tree. Each processor, except those at the leaves, evaluates its assigned position by generating the successors and queuing them for parallel assignment to the slave processors underneath it. Each leaf processor on the other hand evaluates its assigned position by executing the sequential $\alpha\beta$ algorithm. The synchronization is done at each interior processor as when it receives responses from its slaves, it updates its window and notifies the working slaves about the changed window. Thus for a k-level deep processor tree the nodes in the first k levels of the tree can be searched in parallel.

Principal Variation Splitting

At any node on the *principal variation*, the first branch should be searched sequentially before the remaining branches are searched in parallel. This observation lead to the *PV-Split* [MC82] algorithm which divides the nodes along the principal variation in a depth-first order. So once the left subtree in a PV-node is fully searched, the other subtrees rooted at that node are searched in parallel using tree splitting. When search of all subtrees under this PV-node is complete, the evaluated score is returned to the PV-node above it, and so on. Thus only one node's subtrees are being searched in parallel at a single time and all the searching processors must

synchronize at the current splitting node before the algorithm can back up the score to the node above it.

Dynamic PV Splitting

PV-Split algorithm is particularly susceptible to synchronization overheads as all the processors have to synchronize at each splitting nodes. Schaeffer's *Dynamic PV-Split* [Sch89] addresses this issue by allowing dynamic processor trees. The idle processors at a synchronization point can by dynamically reassigned to other busy processors, each of which run the PV-Split algorithm.

Enhanced PV Splitting

Enhanced PV-Split (EPVS) [HSN89] by Hyatt uses a different type of dynamic allocation to make use of the idle processors. It keeps track to when a processor finishes its job and becomes idle. When this happens all the other processors are immediately stopped. The tree is split again two plies deeper down the first remaining node and all the processors start working with these smaller subtrees. The transposition table ensures that the previously computed nodes are not repeated.

Dynamic Tree Splitting

Hyatt's Dynamic Tree Splitting (DTS) [Hya97] eyed to eliminate the situation in which a processor becomes idle with no moves left to search, while the other processors are busy searching. When a processor becomes idle it broadcasts message to the other processor that it has finished searching. In response, the busy processors stores their tree-state data to a shared memory location for the idle processor to examine. The idle processor then establishes a split point and the busy processor copies the complete tree state to a shared memory area and both of them search in parallel. The first choice for a split point is the lowest type 3 nodes or the lowest type 1 node that has its first successor completely searched. Failing this,

a unsearched type 1 node is tried and finally a type 2 node for which more than one move have been completely evaluated is searched.DTS is designed for a shared memory multiprocessor architecture and so all the communication cost is assumed as zero.

Young Brothers Wait Concept

Feldmann et. al introduced the idea of Young Brothers Wait Concept (YBWC) [FMMV89], which states that the search of the younger brothers (all but the leftmost successor) has to wait until the eldest brother (the leftmost successor) is completely evaluated. This idea is similar to the idea of PV if we consider the type 1 nodes; YBWC, however, extends this idea to any node within the game tree. Feldmann's Ph.D thesis [Fel93] includes a few variants by ignoring YBWC rules at type 3 nodes (YBWC-1-2), by searching only 'promising' successors at the type 2 nodes (YBWC*) and combining both of them as YBWC-1-2*. It was implemented in a network of 256 node De-Bruijn connected transputers for the chess program Zugwang.

Dynamic Multiple PV Splitting

Instead of a set of single type 1 nodes at each ply constituting the PV, Dynamic Multiple Principal Variation Splitting (DM-PVSplit) [MGRY95] brought in the idea of PV set. The PV set is the set of most promising nodes at each ply, which comes from the observation that the best observation almost always comes from a small set. The root is by default a member of the PV set. At subsequent levels, nodes are part of the PV set if the parent is a member of the PV set, and they are generated by the first k candidate moves in the move list of the parent. By selecting one candidate for each depth for the set DM-PVSplit generalizes into simple PV-Split.

2.4.4 Asynchronous Search Algorithms

All the algorithms discussed so far have one glaring issue in the form of numerous global synchronization points along the search path. Still asynchronous approaches have not received popularity among the game tree searchers due to the need of quickly backing up the best score to the root to help pruning in the other branches. Only two existing works can be found that focuses on asynchronously searching a distributed game tree.

UIDPABS

Newborn first attempted to asynchronously search a game tree instead of synchronizing at the root. In his Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search [New88], all processors start at the root node and carry out the first two iterations based on which moves are ordered and a narrow window is set. The generated and ordered moves from the root position are then partitioned among the processors, and each processor searches its selected subset of moves iterative deepeningly. The search is unsynchronized among the processors and as such some of the processors may evaluate upto larger depths than others. Initially each processor uses the same window, but as the search progresses some of the processors may have changed their windows based on the search results of their moves. The search terminates once a predetermined time limit has been reached and each processor sends its principal variation and score from the last finished iteration to the master, which then determines the best move.

APHID

Brockington's Asynchronous Parallel Hierarchical Iterative Deepening [BS00] in contrast uses a hierarchical processor tree. For a d-ply search, the master is responsible for the top d'-ply of the tree while the remaining $(d-d')$-ply are searched in parallel by the slaves. Thus the master essentially handles a truncated game-tree, all the

leaves of which are divided equally among the slave processors. Each slave processor continually searches these nodes deeper and deeper never synchronizing with any other sibling processors. The master processor, however, repeatedly search the truncated game-tree to get the latest search result as they are generated by the slave processors.

2.5 Summary

With numerous enhancements already made to the $\alpha\beta$ pruning we almost harnessed all the necessary leverage a single processor can provide. So it seemed natural when parallelism was used to speed up the search technique. We have discussed the notable synchronous and asynchronous parallel search techniques in this chapter. Both of these approaches have their advantages and disadvantages. While the synchronous techniques suffer from the numerous synchronization points, the asynchronous techniques search redundant nodes. However, the asynchronous techniques are more suitable for a loosely coupled system, the types of which are becoming available at larger scale with the advent of Grid technologies.

Chapter 3

Grid Technologies and ChessGrid

3.1 Grid Concept

Grid concept and technologies were initially developed to enable coordinated resource sharing and problem solving within scientific collaborations. The nature of resource sharing is not limited to only file exchange but rather direct access to computers, software, data, and other resources as is required by a range of collaborative computing and problem solving environments. [FKT01] As Foster and Kesselman point out– 'A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities'. [FK04] It will enable software applications to integrate instruments, displays and computational and information resources that are managed by diverse organizations in widespread locations.

One of the main ideas of the Grid is to make computational resources available the same way as electricity is available from the power grid. When we plug in an appliance to the electric power grid infrastructure all we are interested in is getting the electric power. The generators that are actually the source of this power

are remarkably invisible to us. The vision of Grid is to provide a similarly pervasive computational resource grid in which a user can plug in and submit a job which can use diverse computational resources from heterogenous sources. Foster provides a checklist of the minimum properties of a Grid system [Fos02],

- A Grid integrates and coordinates resources that are owned by different companies or under the control of different administrative units and at the same time addresses the issues of security, policy, payment, membership, and so forth that arise in these settings.
- A Grid uses standard, open, general-purpose protocols and interfaces that address such fundamental issues as authentication, authorization, resource discovery and resource access.
- A Grid delivers nontrivial quality of service– relating, e.g., to response time, throughput, availability, security, et cetera.

3.2 Evolution of Grid Technologies

Grid technologies provide mechanisms for sharing and coordinating diverse geographically and organizationally distributed resources so as to create virtual computing system that are sufficiently integrated to deliver desired quality of service. They have come a long way since the term "Grid" was coined in the mid-1990s. Since the advent of Globus Toolkit version 2 (GT2) it has been the de-facto standard for building grids. It defined and implemented protocols and provided a set of tools for application programming (APIs) and system development kits (SDKs). It also included solutions to common issues like authentication, resource discovery and resource access. [FKT01]

The year 2003 saw the emergence of the OGSA-based [FKNT02] (Open Grid Services Architecture, Section 3.3.1) GT3.0. It extends GT2 concepts and technologies to a service-oriented architecture in which an extensible set of *Grid services*

can be aggregated in various ways to meet the needs of *virtual organizations.* By defining a core set of standard interfaces and behaviours it provides a framework within which one can develop a wide range of interoperable, portable services.

In the year 2005 Globus Alliance has released GT4.0 which features a new implementation of the Web Services Resource Framework (WSRF) and the Web Service Notification (WSN) standards. It also provides an API for building stateful Web services targeted to distributed heterogeneous computing environments. However, the focus in this thesis will be on GT3.0 since our algorithm is designed on top of GT3 core.

3.3 Globus & Grid

3.3.1 Open Grid Servicew Architecture

OGSA [FKNT02] represents an evolution towards a service-oriented Grid system architecture based on Web services concepts and technologies. A *service-oriented architecture* (SOA) is essentially a collection of loosely coupled services communicating with each other. A *service* is a unit of work (*operation*) performed by a service provider to achieve desired end results for a service consumer which communicates with the service by a sequence of specific messages. In SOA all entities are services and thus any operation visible to the architecture are actually result of message exchange.

Unlike popular belief *Web service* is not a distributed object. [Vog03] It can be defined as a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP (or REST) messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [Gro04]

To support basic Grid behaviours, a set of WSDL interfaces and associated semantics are provided in OGSA by the Open Grid Service Infrastructure [TCF+03] (OGSI). In essence, OGSI refines and extends the idea of Web service to define mechanisms for discovering, creating, naming, managing lifetime, monitoring, grouping, and exchanging information among entities called *Grid services*. Thus a Grid service description consists of the OGSI-extended WSDL that defines its interfaces and associated semantics. A Grid service instance is an addressable, potentially stateful, and potentially transient instantiation of this description.

3.3.2 Grid Service Components

Naming

Each grid service instance is named, globally and for all time, by one or more *Grid Service Handles* (GSH). However, a GSH is merely a name in the form of URI and to allow a client to effectively communicate with the service instance, GSH must be resolved to a *Grid Service Reference* (GSR). While a GSH is valid for the entire lifetime of the grid service instance, GSR is not a permanent pointer to it and may become invalid for various reasons. GT3 provides a mechanism called HandleResolver to support client resolution of a GSH into a GSR.

Service Data

Service Data is one of the main improvements grid services introduce with respect to plain web services. Service Data allows us to easily include a structured collection of data to any service, which can then be accessed directly through its interface. It is a mechanism to expose a service instance's state-specific data to the service requestors. These requestors are able to query, update, and change these Service Data Elements (SDE). SDE can also be associated with callback notification to get notified whenever its value changes. Every Grid service instance also contain a few standard service data by default as part of the OGSI specification.

Notification

Notification is a core feature to grid service which is closely related to service data. It is a mechanism that allows a *Notification Source* to deliver a message to a notification subscriber (also known as a *Notification Sink*). A Notification Source is a grid service instance that sends notifications, whereas a Notification Sink is a grid service instance or a client that receives it. Notifications use a service data concept behind the scenes. When a service instance wants to receive a notification associated to a particular SDE, the service sends a subscription request to the Notification Source. This subscription request causes the creation of a grid service instance called a *subscription*, which enables the Notification Source to notify the requestor of subsequent changes to the target instances service data.

Life Cycle

The OGSI specification defines the life cycle of any Grid service instance to be "demarcated by the creation and destruction of that service instance". The actual mechanism by which the service instance is created or destroyed depends on the specific hosting environment, however, a collection of related interfaces (*portTypes*) are defined in the specification so that this can be achieved in a similar way.

Grid services solve the *stateless* problem of web services by using a *factory*/instance approach. A factory is a mechanism that can be used by a client to create another Grid service instance. A client can invoke a create operation on a factory thereby receiving a locator to the newly created service, whereas destroying the service may be done by invoking a method on the service instance itself. A service instance may also be destroyed via a *soft-state* approach, where the client registers interest in a service for a specific period of time and when the period expires the service is terminated if no reaffirmation of interest is made from any client.

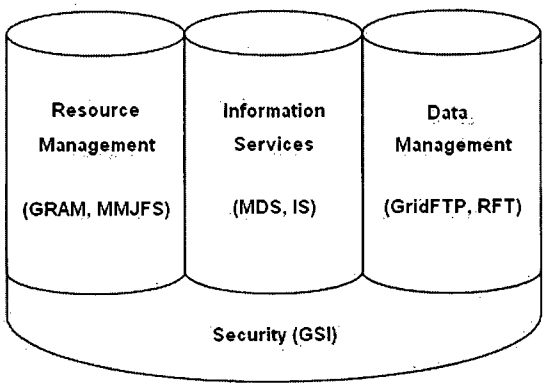


Figure 3.1: Key areas of Grid computing

3.3.3 Globus Toolkit 3.0

Globus Toolkit is an open source software product that has been developed by Globus Alliance to provide middleware services and libraries for the construction of Grid applications. GT3 provides a complete implementation of OGSI in the form of OGSI-compliant services for service discovery, job submission and monitoring, reliable file transfer, and so forth.

The composition of the Globus Toolkit can be pictured as three pillars (Figure 3.1), each representing a primary component of the toolkit based on a common foundation of security. At least one component from each pillar should be included in most Grid implementations. The core architecture [SG03] of GT3 implementation is given in Figure 3.2, where the core components are represented in gray background. Together these components provide the essential building blocks for Grid services.

Core Components

The core components of GT3, as shown in Figure 3.2, contains the basic infrastructure needed for building grid services. It is comprised of the following major

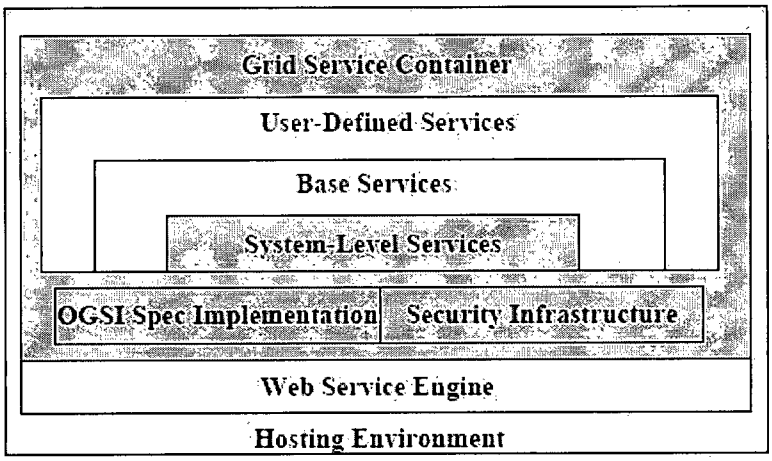


Figure 3.2: GT3 core architecture

subcomponents:

OGSI Spec Implementation: The OGSI Spec Implementation is a set of primitives that provides implementations for all OGSI specified interfaces, as well as APIs and tools for creation, discovery and management of the OGSI compliant services.

Grid Security Infrastructure: The Globus Toolkit uses the Grid Security Infrastructure (GSI) for enabling secure authentication and communication over an open network. It provides SOAP as well as transport level message protection, end-to-end mutual authentication, and single sign-on service authorization.

System level services: GT3 Core also contains some infrastructure level run-time services (*Admin*, *Logging* and *Management* services) that are generic enough to be used by all other Grid services. These so called System-Level Services are built on top of the OGSI Reference Implementation as well as the Grid Security Infrastructure.

Grid Service Container: All these services and primitives interact with an ab-

stract OGSI run-time environment called the Grid Service Container. The container shields the application from environment specific run-time settings and controls the lifecycle of services, and the dispatching of remote requests to service instances.

The first two building blocks OGSI Spec Implementation and GSI do not provide any run time services but serve purely as a base for other services.

GT3 Base Services

The base services implement the three pillars mentioned in Figure 3.1 built on top of the GT3 core.

Job Management Services: Grid Resource Allocation and Management (GRAM) defines a layered architecture in which high-level global resource management services are layered on top of local resource allocation services. It simplifies the use of remote systems by providing a set of interfaces for requesting, using and monitoring remote system resources for the execution of a 'job'. With the advent of GT3, GRAM is divided into two big containers: namely, the Master Hosting environment (MHE) and the User Hosting environment (UHE). MHE contains the Master Managed Job Factory Service (MMJFS), which is responsible for exposing the virtual GRAM service to the outside world. It follows the interfaces defined in OGSI by using WSDL, which is based on XML. Job management services provide a client-side command called *managed-job-globusrun* that invokes the MMJFS to submit a job.

Index Services: The Globus Metacomputing Directory Service (MDS) provides the necessary tools to build an information infrastructure for computational grids. Index services are used in monitoring and discovering services and resources in a distributed system or Grid. Each Grid service instance has a particular set of service data associated with it. The essence of the Index

Service is to provide an interface for operations generating, aggregating and querying any service data element from any grid service.

Data Management: Reliable File Transfer (RFT), along with Grid File Transfer Protocol (GridFTP) and Replica Relocation Service (RLS), is part of the Data Management implementation. It provides the interface for reliable file transfers on Grid servers. GridFTP is an FTP-based high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. RFT, also known as multiRFT, on the other hand is an OGSI-compliant service acting as a proxy for the user to drive third party file transfers. While the RLS maintains and provides access to mapping information from logical names for data items to target names.

Other components

As Figure 3.2 suggests, GT3 core itself is dependent on a couple of components. The container encapsulates the interfaces defined by a standard Web Service Engine, which is responsible for implementing XML Messaging. GT3.2 currently uses the Apache Axis project for this component. The Web Service Engine and Grid Service Container are hosted in a Hosting Environment, which implements traditional Web Server functionality such as the transport protocol (e.g. HTTP). GT3.2 ships with lightweight embedded and standalone hosting environments, but it also works with a standard Java Servlet Engine, e.g. Tomcat, and EJB application servers, e.g. JBOSS or Websphere.

3.4 ChessGrid

In ChessGrid the grid container has a user-level service, $\alpha\beta$Service, deployed in it which can be used by a client to play chess. The algorithm and its internal mechanism will be discussed in Chapter 4, while the service's position with respect

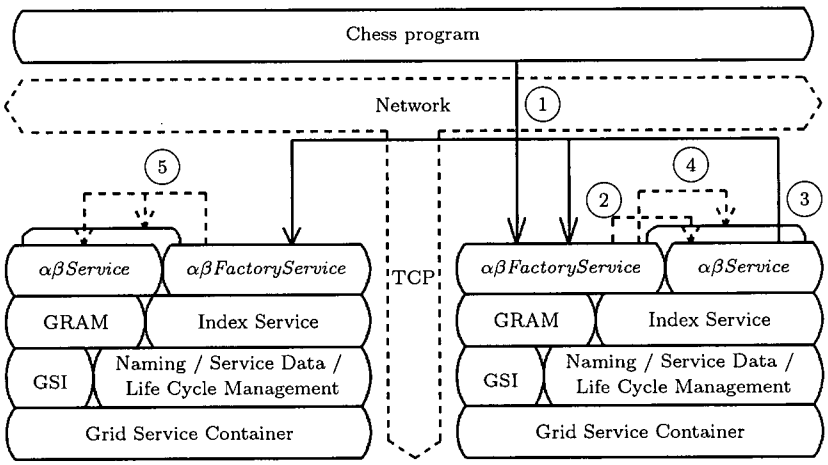


Figure 3.3: ChessGrid architecture and service creation sequence

to the overall grid architecture and the functionalities it uses are described in this section.

3.4.1 Architecture

As the name implies, $\alpha\beta$Service utilizes a service oriented architecture sitting on top of the Globus grid container. It is a transient stateful service and is created by a factory, which is also a user-level service, $\alpha\beta$FactoryService. Both of them are built upon the GT3 base services as shown in Figure 3.3. Communication with the services is made through standard Simple Object Access Protocol (SOAP) and is based on well-defined interfaces that are described using the Web Service Description Language.

$\alpha\beta$FactoryService: It is a factory service that can create new $\alpha\beta$Services, to which a client can send a chess board position for searching the tree underneath it. Itself a persistent service, it implements the OGSI portTypes *NotificationFactory* and *GridService*.

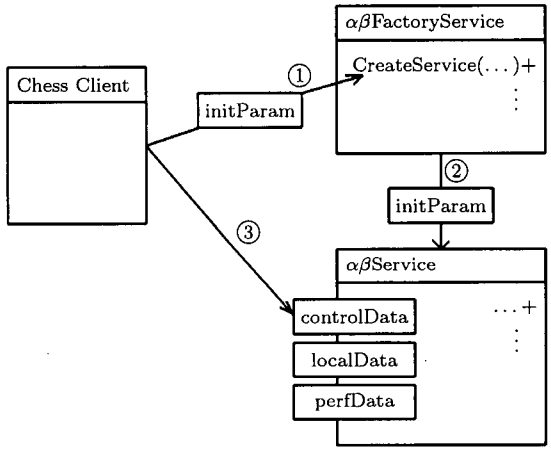


Figure 3.4: Service creation using the factory

αβService: It is a transient stateful service and is created by the αβFactoryService. It provides its functionalities via implementation of an αβPortType which is derived from the OGSI portTypes *NotificationSource*, *NotificationSink* and *GridService*. It is in this service where the game tree searching algorithm is implemented.

3.4.2 Service Creation & Lifetime

A client can request an αβFactoryService to create an αβService via a call to *Factory::CreateService* operation. The factory creates a new αβService instance and returns the Grid Service Handle (GSH) of the newly created service to the client. Now the client can interact with the service using this handle. The process is illustrated in Figure 3.4.

We have used the term 'client' to generalize in this case. It can be the main chess program as well as another service. At the beginning the main chess program requests the αβFactoryservice to create the root αβservice which handles the initial board state. The created service in turn requests the local factory or a factory at a

different site to create child services under it. This request, in the same way, creates new services at the requested location to build a tree of services necessary for the algorithm. The sequence of operation is shown in Figure 3.3.

As an $\alpha\beta$Service is a transient service, it will no longer be available when the time to live expires. It can also be explicitly terminated from a client by a call to *GridService::destroy()*.

3.4.3 Information Sharing

The created services form a parent-child hierarchy. Each set of these parent-child services shares information among them which can be achieved by using three different ways.

Service Data

Each service, as is shown in Figure 3.4, has a number of service data element associated with it. The clients can access the information about their state by a call to *GridService::FindServiceData* operation.

Notification

Notifications are closely related with service data. A client can subscribe to a particular service data element (SDE) in a service, so that it can be notified about any change in the state of that particular SDE. Figure 3.4 shows the sequence of operation that a client follows to subscribe to the LocalData SDE as explained below:

1. *addListener()*: This call subscribes the calling client to a particular SDE (which is specified in the call)

2. *notifyChange()*: Whenever a change happens to that SDE, the service will ask the SDE to notify its subscribers.

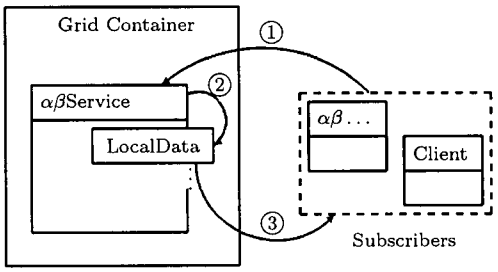


Figure 3.5: Notification mechanism

3. *deliverNotification()*: The SDE notifies the subscribers that a change had happened.

Remote Operation Call

After creating a service, the client receives the GSH of the created service. This can be used to get the portType of the service which then can be used to call the remote operations that the service provides.

3.5 Summary

This chapter was intended to be a brief introduction to Globus Grid. While at it, we have also illustrated how our algorithm exploits grid functionality to build the necessary service hierarchy used by our tree searching algorithm. The various mechanism for data sharing and the implemented portTypes are critical to our algorithm as most of the underlying distributed behaviour is achieved through them.

Chapter 4

$\alpha\beta$Service Algorithm

4.1 Introduction

All synchronous parallel search algorithms have one big issue in common. They all suffer from the numerous synchronization points along the search path. Most of the enhancements made to them are in effect targeted towards minimizing this particular problem. In these enhancements a complex master-slave relationship is often evolved to incorporate the load balancing which is only suitable for tightly coupled systems. Often they use a shared transposition table for improved performance which is also not applicable efficiently in a loosely coupled environment.

Synchronization points are irrelevant for asynchronous algorithm and as such they are free from this problem. UIDPABS, which is also the first asynchronous attempt to search the tree in parallel, achieves reasonable performance when the move ordering is poor. But when the move ordering is good it searches a large number of redundant nodes. In both UIDPABS and APHID, the processors carry out their allocated search not only asynchronously but also independently without communicating with other processors during the search. However, the main argument against the synchronous algorithms was not the communication but the processors sitting idle for the others to finish at the synchronization points. Another important issue is scalability. UIDPABS cannot distribute the tree among more than b (branching

factor) processors. APHID can achieve that by increasing d' (master's searching depth), but in that case master takes more time to complete its search and also all the slaves will be sending their messages to the master creating a communication bottleneck. APHID also allows the implementation of a hierarchical structure within the processes but it is static and determined by the user before the program is started.

In this chapter we introduce a service oriented algorithm for game tree searching, namely $\alpha\beta Service$. Its asynchronous nature prevents it from idle waits at the synchronization points along the search process. It uses a service oriented architecture which allows it to be used in loosely coupled as well as tightly coupled systems. It does not use a shared transposition table and it uses a simple master-slave relationship between a parent and its child nodes, each handled by a separate service. Also in $\alpha\beta$Service the children, after finishing each iteration, look up to the parent service for any update to the search window. The asynchronism stems from the fact that the child services are not waiting at any point of their search for this update. If no update is available from the parent they start the next iteration right away with the current window. It is also scalable as in case of larger resources it can simply parallelize upto greater depth thereby increasing the total search depth.

4.2 Internal Mechanism of Parallel Chess Algorithm

4.2.1 Distribution of the Game Tree

The algorithm employs a tree of services among which the complete game tree is distributed and each service searches the part of the tree allotted to it. The tree distribution technique is explained in Figure 4.1 for a simple tree with branching factor two. Let T be our sample game tree. At the beginning the service S_0 will be called with the root node and the tree $T(S_0)$ will be formed. S_0 will split the tree at the root and delegate node 1 to another service S_1 while itself processing

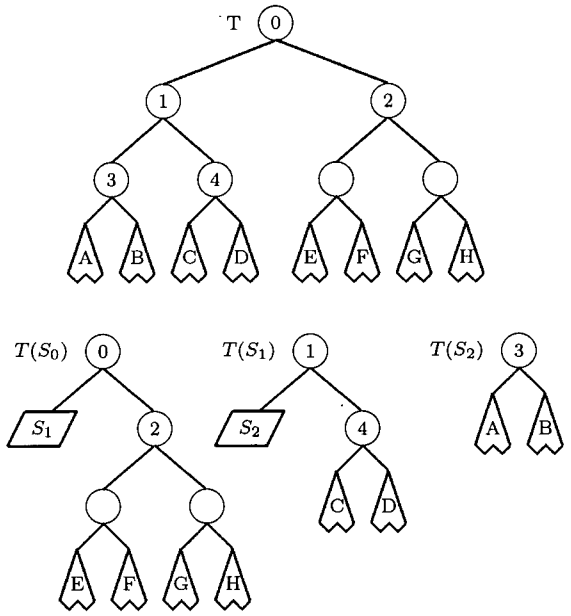


Figure 4.1: A game tree distributed among three services

node 2. The new service S_1 will build the tree $T(S_1)$ and it would again create another service S_2 to process node 3. Each of the created services will carry out an iterative deepening search for the tree it contains. The spawning services (S_0 and S_1) will treat the nodes at which new children are spawned (*spawning nodes* 1 and 3) as *pseudo-leaf*, characteristics of which we will discuss in Section 4.2.2. The nodes 0 and 1 are called *splitting nodes* as the tree under these nodes are split among different services.

4.2.2 Tree Searching Algorithm

Each parallel search is carried out by a separate $\alpha\beta$Service performing an asynchronous game tree search by iterative deepening. An $\alpha\beta$Service starts with a board state and a search window, which is $(-\infty, \infty)$ at the very beginning. Each service first performs a smaller depth search to determine the ordering of the moves at the

root. The returned score is also used to improve the passed window, if possible, by placing a narrow window on it. The depth upto which this search is performed is configurable. However, it is imperative that this initial search takes a very small amount of time and so a two-ply search is made by default.

If the root node for the service is not a *splitting node* then the service carries out a simple iterative deepening search from the root and after each iteration backs up the returned value to the pseudo-leaf in its parent from where it was spawned. However, if the service starts on a *splitting node* then it distributes each possible moves from the root to a separate service. It keeps the last (worst) move for itself while the rest of the moves from the root are delegated to a newly spawned child service. After creating the children, the *spawning nodes* are treated as pseudo-leaves in the parent service. In subsequent iterations these nodes are not expanded, instead evaluation function is called which simply returns the value backed up there by the child service searching that node.

In $\alpha\beta$Service splitting is done at the type 1 (Section 2.4.1) nodes only. Thus a tree of services is formed, the depth of which is controlled by the *parallel search depth* (*parDepth*). So all the type 1 nodes upto *parDepth* level distributes their moves to the child services (Figure 4.2).

The spawned services maintain a child-parent relationship with the spawning service and are connected by notification subscription. Each child service is also responsible for searching the game tree rooted at the node allocated to it. After finishing each iteration it sends a notification to the parent service about the updated score. Thus a good score found after one iteration in any child is backed up to its parent. At the same time the child service also looks up the parent's (α, β) window to see if any changes has occurred to it due to finding a better path somewhere else in the tree. It compares its current window with the parent's window and starts the next iteration.

The services are unsynchronized in the sense that when a parent service

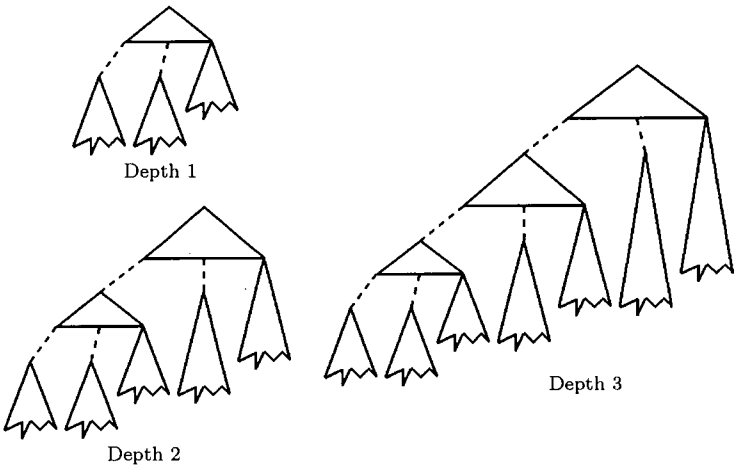


Figure 4.2: Master-slave hierarchy in $\alpha\beta$Service for different parallel Depth

finishes an iteration, it does not wait for its child services to finish before beginning the next. Instead it always treats whatever value that is backed up at the spawning node as its current payoff value. Similarly the child services do not wait for their siblings or try to share their jobs. They just back up the payoff value returned by their deepest completed iteration to corresponding spawning node and start the next iteration. So at the same time the children and parent may or may not be at the same level of iterative deepening.

In Figure 4.3 we illustrate the difference in work allocation between $\alpha\beta$Service and other asynchronous search techniques. The tree splitting locations are marked as x in the figure. For simplicity a perfect ordering is assumed in the figure for $\alpha\beta$Service which parallelizes upto level 3, which is configurable using the parameter *parDepth*.

4.2.3 Service Description

The complete GWSDL for $\alpha\beta$Service can be found in Appendix A. It provides the following service data, defined as part of the $\alpha\beta$portType:

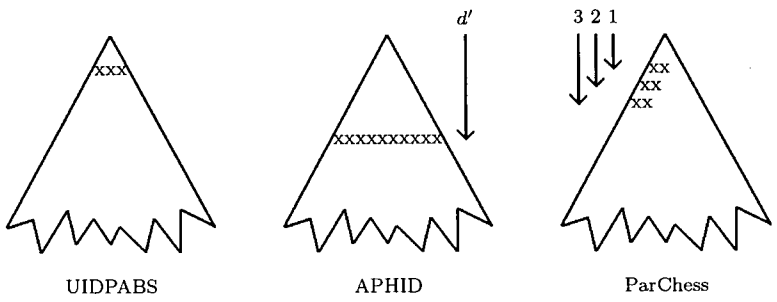


Figure 4.3: Work allocation comparison

ControlData: It includes the parameters needed to customize the search performed by the service. Notable among these are:

- parDepth - Depth upto which parallelism is employed.
- wndDepth - Depth upto which search is done at the first step to determine the initial window.
- $\alpha\beta$ window- Current window at the root of the tree.

LocalData: It holds the search results from the last finished iteration which can be categorized into:

- Identification - Child's ID from the parent and node's ID from the root in Dewey decimal format.
- Search status - Current iterative deepening depth, generated tree depth, etc.
- Search result - Current principal variation, payoff value, etc.

PerfData: It includes the data required for analyzing the performance: e.g. Node counts, search time. tree depth, etc.

The service constructor takes a parameter (*initParam*) which initializes ControlData and the current board state. It also defines the following remote operations

which can be accessed by the portType.

- *setNodeType(type)* - Sets the type of the root node.
- *getAlpha(depth) & getBeta(depth)* - Returns the window at the specified depth from the root.
- *addListener(servicedata, handle)* - Add the service with the passed handle as a subscriber to the specified service data.
- *killChilds()* - Kill all the children under the service.
- *deliverNotification(extensibility)* - Called when any change occurs to the service data it is subscribing to.

4.2.4 Flow of Information

Our algorithm distributes the total game tree over a number of unsynchronized iterative deepening services. However, as the services are running on top of a powerful Grid container, some Grid functionalities are exploited to back-up the score, yet not waiting for the synchronization among these services. The parent-child set at each depth shares some information among them.

Payoff value: When a child service finishes one iteration, the payoff value calculated is sent to the parent to back it up.

Search window: After each iteration the child service looks up its parent's search window and changes its own window accordingly.

Node type: As search gets deeper, a type 1 node can later turn out to be a suboptimal move. In that case a parent changes the node type of its children to update the current tree structure.

The information sharing for a parent-child scenarios is shown in Figure 4.4. Payoff value is part of LocalData and a parent subscribes to this service data of each

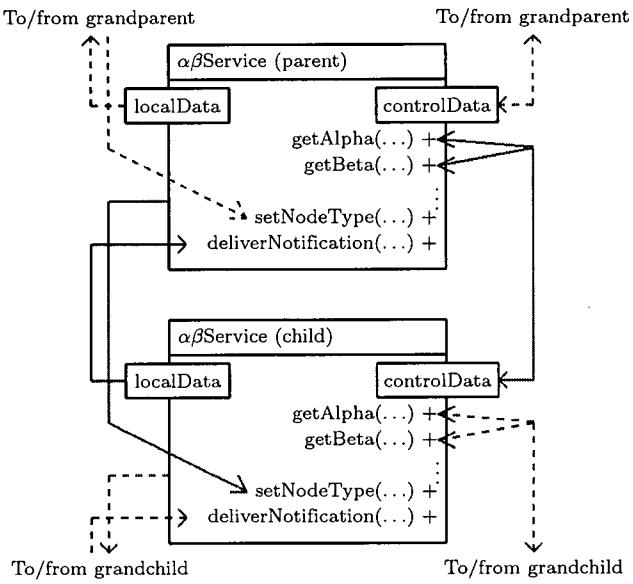


Figure 4.4: Information sharing between the services.

child. So whenever a change happens to it, the parent is notified and it changes the value in the corresponding pseudo-leaf. (α, β) windows of a service's root node is stored in ControlData but child services can spawn from different depths in its game tree. So after each iteration the child service queries its parents search window specifying the depth from which it was spawned. Again, a node type may change over the course of the search. In case of such a change, the parent updates its children's node types.

4.3 Implementation Details

In order to perform a game tree search, aside from the searching algorithm, we also had to design a chess program. However, very few of the ideas are entirely original; many represent elementary chess knowledge and have been used in other chess programs. Sources that have been of particular influence are Slate & Atkin [SA77]

and Hyatt [Hyaa, Hyab]. Some of the key concepts are discussed in the following subsections.

4.3.1 Chess Board Representation

The first important decision in a chess program design is choosing the data structure to represent the chessboard. Back in the early days when computer memory was at a premium, the board was represented in the most compact way. As memories became comparatively cheaper an extended board representation became much more common so as to make the operations like move generation, position evaluation, and so forth more efficient.

Two distinctly different board representation styles can be seen in current chess programs. The most common data structure is generally referred to as the *offset board representation.* It uses an 8×8 array of elements with each square of the chess board mapping to one element of the array. It is, however, common to have extra rows and columns in the array, containing values denoting the edge of the board to speed up move generation.

A newer approach uses a set of *bitmaps* or *bitboards* to represent the chess board. A bitboard is essentially a 64-bit number, of which each bit represents a square on the board. Thus $A1$ is mapped to 0-th bit (LSB), $H1$ to 7-th, $A8$ to 56-th and $H8$ to 63-rd bit(MSB). The construction of a bitboard for white pawns is shown in Figure 4.5. 12 (6 for white, 6 for black) such bitboards are used to represent each type of piece on the board. We also kept a few more special purpose bitboards to minimize OR operations at later stages.

The main advantage of using bitboard lies in the fact that much of the move generation and board evaluation related steps can be performed by trivial bitwise operations. And besides, 64-bit microprocessors are already here and they would offer an immediate performance improvements to bitmap approach as they do the very operations on 64-bit register.

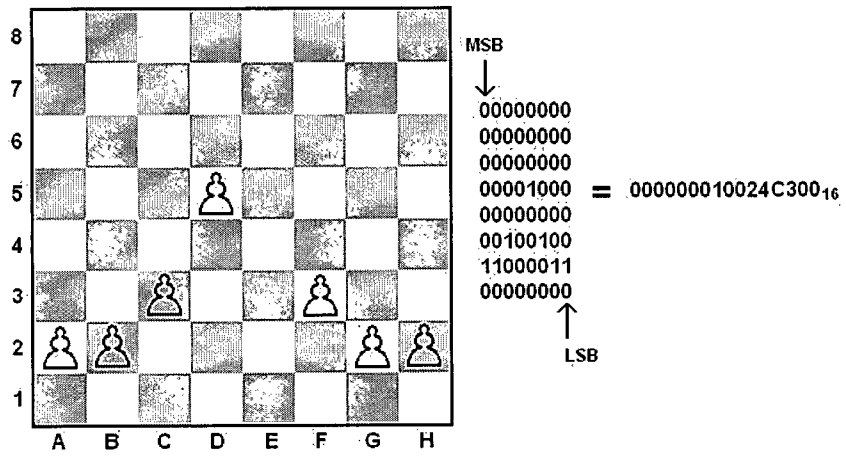


Figure 4.5: Constructing white pawn bitboard from piece positions

4.3.2 Move Generation

Slate & Atkin developed the idea of bitboards and described how to construct and incrementally update an additional set of bitmaps that contained two types of information.

- `attacksFrom[64]` is an array of bitmaps that contain 1 bits set for each square attacked by the piece that is on a particular square.
- `attacksTo[64]` is an array of bitmaps that contain 1 bits set for any square that is occupied by a piece which is attacking the particular square.

Once these two bitmaps are ready, possible moves for a piece can be generated by finding all the squares that have a 1 in the `attacksFrom` bitmap for the square that the piece is placed on. Capture moves can be distinguished by a simple AND operation with all pieces bitmap for the opponent.

Constructing the attack bitmaps (`attacksFrom`) for non-sliding pieces (pawn, knight and king) is trivial since they are independent of the pieces placed in the other squares. So all the squares a given type of non-sliding piece attacks from any square

can be precomputed and be readily used. However, such precomputed bitmaps does not work for the sliding pieces (bishop, rook and queen) as they only attack in the direction of the slide but not beyond the first piece they encounter in that direction. To solve this issue a set of directional masks are used that contain 1 bits on any square that a sliding piece on a square attacks in the specified direction. Each direction mask itself represents the squares that a sliding-piece on a square would attack if there are no blocking pieces in that direction. However, AND-ed with all pieces bitmap it can also identify the location of the first blocking square. Same direction mask for the blocking square is then used to truncate the attacks beyond the blocking piece. This process is repeated for four directions to get the attack bitmap for rook and bishop, and all eight for the queen.

To compute the attacking bitmaps (`attacksTo`), attack bitmaps for each type of piece is computed first which is then AND-ed with that specific type of pieces present on the board. All these results for a square OR-ed together enumerate every square that is attacking the particular square.

4.3.3 Move Representation

Each move is represented by a 32-bit word. It includes the information for source square, destination square, moving piece type and captured piece type, if any. For promotions it includes the type of the promoted piece. There is also separate chunk for special moves like en-passant captures and castling moves.

4.3.4 Making a move

Move generation is used to generate the possible moves from a specific node in the tree. But to traverse the game tree from a parent to a child node, we need to generate the successor position resulting from the selected move. Similarly to go back up the game tree during the traversal we need to restore the previous position by unmaking the move. Each of the two operations require updating two types of

information: i) positional bitboards so as to represent the new board state and ii) helper bitboards and other informations (e.g. attack and attacking bitmaps) so that they conform to the already changed board state.

4.3.5 Evaluation Function

The static evaluation function returns a score for the side to move from the given position. A score is calculated for both sides and the function returns the score for the side on the move minus the score for the side not on the move. The main dilemma with the static evaluation function is that, if they are to be precise they must be sufficiently complex and costly which in effect limits the extend of the search. Another approach can be using an ordinary and cheap termination heuristics and compensate its imprecision with extensive search. Our evaluation function follows the later approach as the factors considered have been chosen because they are relatively quick to calculate.

Pawn Scoring

Each pawn scores 100 points. A side is penalised for having more than one pawn on the same file (doubled pawns), pawns which have no neighbor pawns capable of protecting it from attacks (isolated pawns), blocked pawns and pawn rams. Passed pawns are awarded a bonus that relates to the pawn's rank number. Further bonus is given if there are no enemy pieces in front it. Connected passed pawns are also awarded based on their ranks.

Rook Scoring

Each rook scores 500 points. Rooks are awarded a bonus for king tropism that is based on the minimum of the rank and file distances from the enemy king. If a rook is on an open file or in a semi complete file or is behind a passed pawn it receives a bonus. Bonus is also awarded for two friendly rooks or a rook and queen sharing

the same file.

Bishop & Knight Scoring

Each of them scores 300 points and are awarded bonuses for closeness to the centre of the board and also for closeness to the enemy king. They are also penalized/awarded for blocking self/opponent's center pawns. Bonus points are given for outposts, which is a knight that can't be driven off by an enemy pawn and which is supported by a friendly pawn. A bonus is given for the presence of two bishops and bishop mobility, which is the number of possible moves by the bishop. Also penalize pawns on the same color as the bishop, if one side has only one bishop(bad bishop).

Queen Scoring

Each queen scores 900 points. Queens are awarded points for closeness to the enemy king. A special bonus is awarded for queen in 7th rank with the opponent king stuck at 8th rank. Further bonus is awarded if the 7th rank is also supported by a rook.

Piece Development

At early stages bonus is given for castling or penalized for loosing rights to do so. Unmoved center pawns and bishop and knights staying at rank 1 are penalized for piece development. An early queen movement is penalized if the other pieces are not already developed.

King Safety

If the number of enemy pieces and pawns in the friendly king's board quadrant is greater than the number of friendly pieces and pawns in the same quadrant, the side is penalized. When considering enemy presence in the quadrant a queen is counted as three pieces.

The evaluation function does not detect checkmate. Evaluation of won, drawn of lost positions is left to a function that is called when a position is found in the search from which there are no available moves. A sufficiently big value(99,999) is set as payoff for a won position with the depth at which such a position is discovered being subtracted from this score. This encourages the program to take the shortest sequence of moves to win a game. Similarly, the depth at which lost positions are discovered is added to the value -99,999 to encourage to program to delay the loss for as long as possible.

Checks are also left for the subsequent move by the opponent to take hold of the king. Piece value of king is set to 32,000 so that this move is always deemed unpromising.

4.4 Summary

In this chapter we have described our searching algorithm $\alpha\beta$Service. Although it is asynchronous, information from one service can still reach other services via notification and other mechanisms. It does not employ a shared transposition table and the performance evaluation presented in the next chapter is done without using any transposition table.

Chapter 5

Performance Evaluation

5.1 Introduction

The performance of a sequential algorithm is usually evaluated in terms of its execution time. When evaluating the performance of a parallel algorithm we are often interested in measuring the benefit achieved by the parallelization of the problem, also in terms of the execution time. Some common metrics that are used to measure the performance of a parallel system are introduced in Section 5.2. The methodology used for evaluating the performance of $\alpha\beta$Service is outlined in Section 5.3. Finally in Section 5.4 test results are listed and analyzed.

5.2 Parallel Algorithm Terminology

5.2.1 Defining Speedup and Efficiency

The basic idea of parallel computing is to use several processors to perform a single task. The key issue of measuring the quality of such algorithms is the speedup achieved, especially its dependence on the number of processors used. *Speedup* is defined in terms of the time taken by the best sequential algorithm. Let P be a computational problem with an input size n. Let the best sequential implementation of the problem solves it in time $T(P_n)$. Now if a parallel algorithm solves the problem

in time $T_p(P_n)$ by running it parallely on p processors, then the speedup achieved is given by the equation:

$$\text{speedup} = \frac{T(P_n)}{T_p(P_n)} \tag{5.1}$$

Sometimes, the best sequential algorithm for a targeted problem is unknown and a comparison is made with the parallel algorithm run on single processor. This is defined as *relative speedup* and the former as *absolute speedup* to distinguish the approaches.

$$\text{relative speedup} = \frac{T_1(P_n)}{T_p(P_n)} \tag{5.2}$$

However, a parallel algorithm performs many additional tasks compared to its sequential counterpart. So a third measure of speedup is also seen where a sequential algorithm with the same methodology as the parallel one is used as the baseline to measure speedup. This is defined as the *observed speedup.*

In designing a parallel algorithm, it is more important to make it efficient than to make it asymptotically fast as *efficiency* measures how well the system is utilized.

$$\text{efficiency} = \frac{\text{speedup}}{p} \tag{5.3}$$

So an algorithm that obtains a speedup of $O(\sqrt{n}\log n)$ using $\sqrt{n}$ processors is better than the algorithm that obtains a speedup of $O(\log n)$ using n^2 processors.

Efficiency can also be defined with respect to the measure adopted for the speedup and likewise be called *relative efficiency* or *observed efficiency.* Most of the speedups and efficiencies that will be discussed in this document are observed unless otherwise specified.

5.2.2 Limiting Factors: Overheads

Only an ideal parallel system can deliver a speedup of p when using p number of processors. In practice, however, the ideal behaviour is not obtained as the parallel

algorithm has to spend a considerable amount of time to perform additional tasks to achieve the parallelism. All causes of nonoptimal efficiency in a parallel system are collectively referred to as *overhead* due to parallel processing.

Thus, in effect, parallel algorithms take more time to acquire the same result. The additional amount of processing that a parallel algorithm needs to perform is expressed with *total overhead* which is defined as,

$$\text{total overhead} = \frac{T_p(P_n) \times p - T(P_n)}{T(P_n)} \tag{5.4}$$

For game tree searching, a second factor influencing the speedup can be measured using the number of nodes generated by the parallel algorithm. If the parallel algorithm searches $N_p(P_n)$ compared to $N_1(P_n)$ by the sequential one then the *search overhead* can be expressed as,

$$\text{search overhead} = \frac{N_p(P_n)}{N_1(P_n)} \tag{5.5}$$

Search overhead is very vital for the asynchronous algorithms as the parallely running processors do not synchronize among themselves to minimize the number of redundant nodes searched. Synchronous algorithms, on the other hand, must wait at the synchronization points until all the concurrently running processors agree on the next status of the search. This leads to the *synchronization overhead*, which is the amount of time lost due to the idle processors.

$$\text{synchronization overhead} = \frac{\text{time spent for at synchronization points}}{T(P_n)} \tag{5.6}$$

Needless to point out that the asynchronous algorithms do not suffer from this overhead as the processors do not sit idle waiting for the others at any point in the search (except at the end of the search).

The *parallelization overhead* is the amount of time spent by the algorithm to achieve and later maintain the parallel behaviour. This overhead is problem specific as it includes the measures adopted to ensure the parallelism. It includes the time spent in updating the complex data structures used to keep track of the parallel

job. It also includes the *communication overhead* which is the amount of time the parallel algorithm spends by sending and receiving messages.

5.3 Evaluation Methodology

5.3.1 Testing Environment

A Grid testbed is created for testing the algorithm using four machines in the Netbed cluster and two desktop PCs. Netbed cluster comprises of a set of 24 IBM xSeries 306 servers, each of which has the following configuration,

- Intel®Pentium®4 CPU 3.20GHz with 1024KB cache size.
- 1GB main memory and simple-swap SATA 80GB internal storage.
- 1000 Mbps Full Duple Intel®PRO/1000 NIC.
- Fedora Core 3 operating system.

Each of the desktop pc has the following configuration,

- Intel®Pentium®4 CPU 2.80GHz with 512KB cache size.
- 512 MB main memory and 40GB internal storage.
- 1000 Mbps Full Duple Intel®PRO/1000 NIC
- RedHat 9 operating system (kernel-2.4.20-8).

The four cluster machines work as servers, while the two desktop machines work as both client and servers, Figure 5.1. Each machine has GT3 installed in it and has a Globus Container running with $\alpha\beta$Service deployed in it.

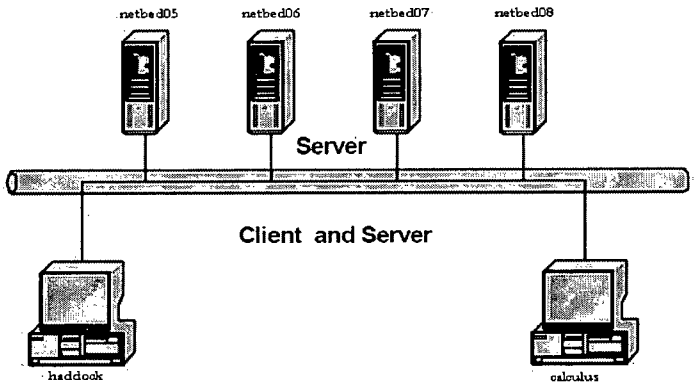


Figure 5.1: Grid Testbed Environment

5.3.2 Generating the Test Cases

Chess game trees have an average branching factor of 35 to 40. Thus they need six services (on average) per machine in our testing environment, which severely affects the performance of the algorithm. In order to limit the number of services per machine to less than or equal to three (with parDepth upto 2 for six processors), new test cases are generated with a maximum branching factor of six. But instead of artificially creating the trees, we have taken some well known test positions and modified them according to our requirement.

The Bratko-Kopec Test [BK82] was designed by Ivan Bratko and Danny Kopec in 1982 to evaluate human or machine chess ability. This test has been a standard for nearly 20 years in computer chess to reliably rate an algorithm's playing strength. The complete test includes 24 positions numbered as 1...24 and each considered either tactical (T) or lever/positional (L). The positions where the lack of chess knowledge can be compensated by calculation are considered tactical, while lever positions are those where it cannot be compensated. A level of difficulty from 1 to 4 has been assigned to each position as well, 5.1.

Difficulty Level	Tactical	Lever
1	1, 12	-
2	10, 14, 15, 21	2, 6, 9, 11, 13, 17, 20, 24
3	5, 16, 19	3, 4, 8
4	7, 18, 22	23

Table 5.1: Summary of Bratko Kopec Test Positions

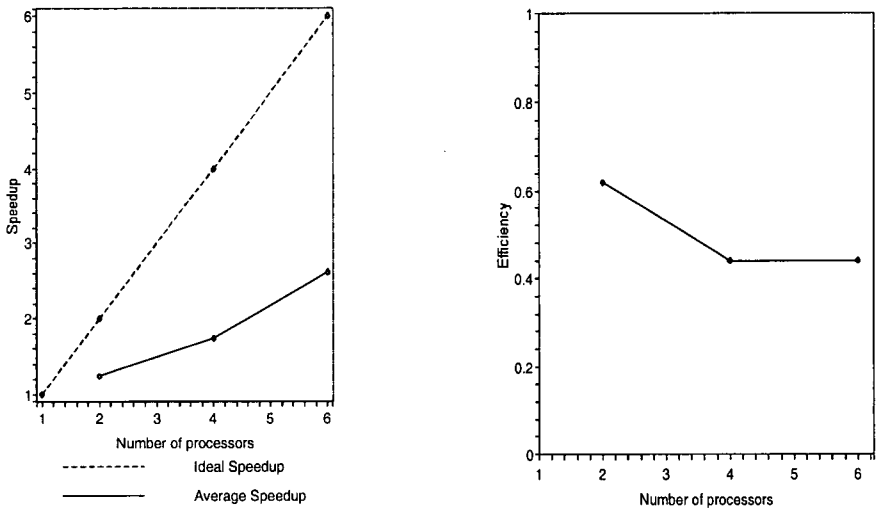
5.4 Experimental Results

While analyzing our algorithm difficulty arose from the fact that due to asynchronous nature of the algorithm the problem size at different stage were not similar to its sequential implementation. The concurrently running services were also not searching at the same ply depth at the same time. So, to meaningfully evaluate the fixed size model that we are using, synchronization is added at the end of the search. The parent service waits for the children to finish after completing its allocated partition of the tree.

As part of the simulation, ten 12-ply depth search was conducted on each of the 24 positions of the test set for 1, 2, 4 and 6 processors. They are then averaged to calculate the speedup and overhead of each case which are then averaged again for the overall speedup and overhead. This method was used to make sure that each of the test case receives equal weight and an excellent performance in one of them does not inflate the overall performance. These results are reported in Table 5.2.

p	Speedup	Efficiency	Total Overhead	Search Overhead
1	-	-	-	-
2	1.24	0.62	0.63	1.48
4	1.74	0.44	1.45	1.53
6	2.61	0.44	1.49	1.54

Table 5.2: Average test results for the test set



(a) Average Speedup (b) Average Efficiency

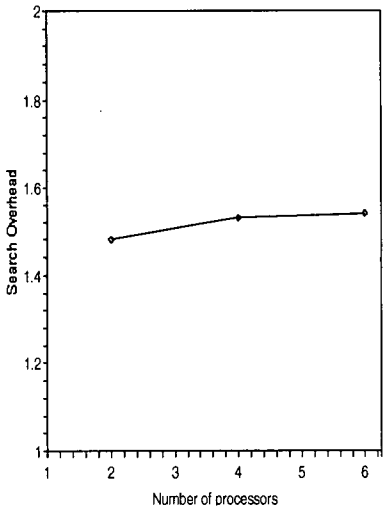
Figure 5.2: Performance curves - Speedup and Efficiency

5.5 Discussion

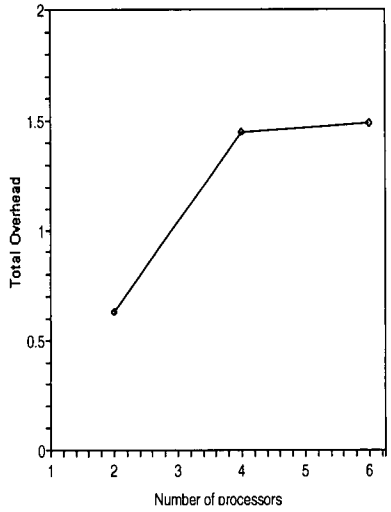
Figure 5.2(a) shows the observed speedup obtained from the test results. We can see that it is nearly linear with respect to the number of processors used. Although the overall speedup for p processors is less than $p/2$, it can be due to the following reasons:

- Our algorithm does not use any transposition table.
- All the test positions are searched upto the same 12-ply depth. However, the test positions used generated a widely varying sized game tree and for some smaller sized trees no speedup was obtained using the parallel search.

Efficiency curve in Figure 5.2(b) shows a gradual descent as the number of processor is increased. This can be compared to the two overhead curves, Figure 5.3(a) & 5.3(b), which validate the decrease in the efficiency. Both the 2-processor



(a) Search Overhead



(b) Total Overhead

Figure 5.3: Overhead curves

and 4-processor trials use $parDepth = 1$, while 6-processor cases are tried for both $parDepth = 1$ and $parDepth = 2$. However, Significant speedup was not obtained for greater service level parallelism which is not unusual considering it is a compute-intensive problem. An increase in slope of the speedup curve is also noticeable in Figure 5.2(a) when six processors are used. This further corroborates the fact that a single service per machine generates better performance compared to multiple services per machine. However, it is interesting to note that when the same number of services are distributed over a lesser number of processors the search overhead is found to be less. It can possibly result due to the fact that some of the concurrently running services in the same processor actually had to wait due to process switching.

Chapter 6

Conclusion

6.1 Summary

We have developed a service oriented asynchronous algorithm that exploits Globus Toolkit's existing OGSA platform services to search a game tree parallely. The algorithm exhibits a clearly linear speedup upto six processors. But due to the small number of processors available for the experimentation it cannot be stressed that the algorithm will scale the same way for a large number of processors. Analysis have shown that the search size is an important factor in determining the observed speedup. With increased number of machines available, the algorithm will be able to partition the tree into more parts. Another positive side is that the search overhead curve shows that the number of redundant nodes searched also grows gradually. Thus we can be optimistic that the algorithm will scale well with larger number of nodes. As the number of processors becomes larger the bottleneck at the master node in APHID can become a major issue. So a hierarchical approach like $\alpha\beta$Service may be considered instead of the static master-slave configuration used by APHID.

The best known speedup for sequential parallel search algorithm is shown by Feldmann's YBWC. It generates a speedup of 142.62 on 256 processors used. However, their analysis was made by averaging all the test results together to generate the combined time, which then was used to calculate the speedup. APHID builds

a game independent parallel search library that can be easily inserted into legacy sequential search algorithms. Their search result for Keyano (average branching factor 10) shows a speedup of 5.74 with eight processors for a 15-ply deep search in a fixed-depth shared memory environment. For 64 processors in the same environment it shows a speedup of 37.44. Both of them use a transposition table for better performance. $\alpha\beta$Service demonstrates a speedup of 2.61 with six machines without using any transposition table. Further research is needed to correctly predict whether $\alpha\beta$Service can match the performance of the other two algorithms for massively distributed systems.

6.2 Future Study

Perhaps the most obvious extension to the current work is to carry out the experimentation for a larger number of processors. One possible direction for that is to make use of the schedulers to achieve the service level parallelism. The Index Service can be used to register and later discover the services in such a system.

Although $\alpha\beta$Service is targeted for a loosely coupled system, the testing environment with a gigabit network connection did not explore the performance of the algorithm in such a system. One fascinating extension can be to couple geographically distributed Grid resources together for the algorithm. Another approach can be to simulate WAN charactersitics like latency, packet-loss and bandwidth constraints in the system and evaluate the algorithm under these limitations.

The underlying basic $\alpha\beta$ pruning in the algorithm can also be targeted for some improvements. the most notable among them is the use of a transposition table which can be independent or shared among the services. As shared memory cannot be used for the system, a shared transposition table should try to minimize the number of communication.

The role of the services over their lifetime needs to be changed in order to allow continuous game play. The functionality of the root service can be extended

so that when the opponent makes a move, it can find the service among its children/grandchildren that handles the new position and transfer control to it. This, in effect, selects a new service as the root; thereby retaining previously computed part in the new game tree.

As there can be diverse computational resources available in the Grid, some machine specific information can be included as service data to the factory, which can then be queried to determine the number of services created at that site.

Globus Toolkit also incorporates security measures for enabling secure authentication and communication over an open network which is completely overlooked in the course of this thesis.

Bibliography

[Bau78] Gerard M. Baudet. *The Design and Analysis of Algorithms for Asynchronous Multiprocessors.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, April 1978. 20

[BK82] Ivan Bratko and Danny Kope. *The Bratko-Kopec experiment: a comparison of human and computer performance in Chess.*, pages 57–72. Pergsmon Press, Oxford, 1982. 57

[Bro96] Mark G. Brockington. A taxonomy of parallel game-tree search algorithms. *ICCA Journal*, 19(3):162–174, 1996. 20

[Bro98] Mark G. Brockington. *Asynchronous Parallel Game-Tree Search.* PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Canada, 1998. 20

[Bru63] A. L. Brudno. Bounds and valuations for shortening and scanning of variations. *Problemy Kibernetiki (in Russian)*, 10:141–150, 1963. 12

[BS00] Mark G. Brockington and Jonathan Schaeffer. Aphid: Asynchronous parallel game-tree search. *Journal of Parallel and Distributed Computing*, 60(2):247–273, 2000. 3, 24

[CJhH02] Murray S. Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002. 20

[Ebe86] Carl Ebeling. *All the Right Moves: A VLSI Architecture for Chess.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, April 1986. 20

[Fel93] Rainer Feldmann. *Game Tree Search on Massively Parallel Systems.* PhD thesis, University of Paderborn, Paderborn, Germany, May 1993. 23

[FF82] Raphael A. Finkel and John P. Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19:89–106, 1982. 18, 20, 21

[FK04] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers Inc., 2 edition, 2004. 26

[FKNT02] Ian Foster, Carl Kesselman, J. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, June 22, 2002. 3, 27, 28

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications and High Performance Computing*, 15(3):200–222, 2001. 3, 26, 27

[FMMV89] Rainer Feldmann, B. Monien, P. Mysliwietz, and O Vornberger. Distributed game-tree search. *ICCA Journal*, 12(2):65–73, 1989. 23

[Fos02] Ian Foster. What is the grid? a three point checklist. *GRIDToday*, July 20, 2002. 27

[Gro04] W3C Working Group. Web services architecture. February 11, 2004. 28

[hH90] Feng hsiung Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorighmic and Architectural Study.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, February 1990. 20

[HSN89] Robert M. Hyatt, Bruce W. Suter, and Harry L. Nelson. A parallel alpha/beta tree searching algorithm. *Parallel Computing*, 10:299–308, 1989. 22

[Hyaa] Robert M. Hyatt. Chess board representations. Online Technical Papers. 47

[Hyab] Robert M. Hyatt. Rotated bitmaps. Online Technical Papers. 47

[Hya97] Robert M. Hyatt. The dynamic tree splitting parallel search algorithm. *ICCA Journal*, 20(1):3–19, 1997. 22

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975. 12, 19

[MC82] T. A. Marshland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, December 1982. 21

[MGRY95] T. A. Marsland, Yaoqing Gao, A. Reinefeld, and A. Yonezawa. Multiple principal variation splitting search. In *High Performance Computing Symposium, HPCS '95*, pages 292–303, July 1995. 23

[New88] Monroe Newborn. Unsynchronized iterative deepening parallel alpha-beta search. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 10(5):687–694, Sept 1988. 3, 24

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers Inc., 1980. vii, 10, 11

[NSS58] Alan Newell, J. C. Shaw, and H. A. Simon. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2(4):320–355, 1958. 12

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach.* Pearson Education Inc., 2nd edition, 2003. 6

[SA77] David J. Slate and Lawrence R. Atkin. *CHESS 4.5- The Northwestern University chess Program*, pages 82–118. New York: Springer-Verlag, 1977. 2, 16, 18, 46

[Sch89] Jonathan Schaeffer. Distributed game-tree searching. *Journal of Parallel and Distributed Computing*, 6(1):90–114, February 1989. 22

[SG03] Thomas Sandholm and Jarek Gawor. Globus toolkit 3 core a grid service container framework, July 2, 2003. 31

[Sha50] Claude E. Shannon. Programming a computer for playing chess. *Phylosophical Magazine*, 41:256–275, 1950. 1, 9

[SP96] Jonathan Schaeffer and Aske Plaat. New advances in alpha-beta searching. In *ACM Conference on Computer Science*, pages 124–130, 1996. 15

[TCE01] *The Columbia Encyclopedia.* Columbia University Press, 6th edition, 2001. 1

[TCF+03] Steven Tuecke, K. Czajkowski, Ian Foster, et al. Open grid services infrastructure (ogsi) version 1.0. *Global Grid Forum Draft Recommendation*, June 27, 2003. 29

[Tur53] Alan M. Turing. *Digital Computers Applied to Games*, pages 286–310. Pitman, 1953. 2

[vNM44] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior.* Princeton University Press, 1944. 5, 6

[Vog03] Werner Vogels. Web services are not distributed objects. *Internet Computing*, 7(6):59–66, 2003. 28

[WIK05] Wikipedia, the free encyclopedia, 2005. accessed on January, 2005. 1

Appendix A

ChessService GWSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ChessService"
    targetNamespace="http://onindyo.sukanta.com/ChessService"
    xmlns:tns="http://onindyo.sukanta.com/ChessService"
    xmlns:ogsi="http://www.gridforum.org/namespaces/2003/03/OGSI"
    xmlns:gwsdl="http://www.gridforum.org/namespaces/
        2003/03/gridWSDLExtensions"
    xmlns:sd="http://www.gridforum.org/namespaces/2003/03/serviceData"
    xmlns:data="http://onindyo.sukanta.com/ChessService/ControlData
    http://onindyo.sukanta.com/ChessService/LocalData
    http://onindyo.sukanta.com/ChessService/InitData"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

<import location="../../ogsi/ogsi.gwsdl"
    namespace="http://www.gridforum.org/namespaces/2003/03/OGSI"/>
<import location="InitDataType.xsd"
    namespace="http://onindyo.sukanta.com/ChessService/InitData"/>
```

```
<import location="ControlDataType.xsd"
    namespace="http://onindyo.sukanta.com/ChessService/ControlData"/>
<import location="LocalDataType.xsd"
    namespace="http://onindyo.sukanta.com/ChessService/LocalData"/>
<import location="PerfDataType.xsd"
    namespace="http://onindyo.sukanta.com/ChessService/PerfData"/>

<types>
<xsd:schema targetNamespace="http://onindyo.sukanta.com/ChessService"
    attributeFormDefault="qualified"
    elementFormDefault="qualified"
    xmlns="http://www.w3.org/2001/XMLSchema">

    <!--BEGIN ELEMENT DEFINITIONS - DO NOT MODIFY THIS BLOCK!!! -->
    <xsd:element name="deliverNotification">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="arg1" type="ogsi:ExtensibilityType"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="deliverNotificationResponse">
        <xsd:complexType/>
    </xsd:element>
    <xsd:element name="killChilds">
        <xsd:complexType/>
    </xsd:element>
    <xsd:element name="killChildsResponse">
```

```
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="value" type="xsd:boolean"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="addListener">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="arg1" type="xsd:string"/>
                <xsd:element name="arg2" type="ogsi:HandleType"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="addListenerResponse">
        <xsd:complexType/>
    </xsd:element>
    <xsd:element name="setNodeType">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="arg1" type="xsd:byte"/>
                <xsd:element name="arg2" type="xsd:boolean"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="setNodeTypeResponse">
        <xsd:complexType/>
    </xsd:element>
```

```
<xsd:element name="getAlpha">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="arg1" type="xsd:byte"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="getAlphaResponse">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="arg1" type="xsd:int"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="getBeta">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="arg1" type="xsd:byte"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="getBetaResponse">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="arg1" type="xsd:int"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

```
    <!--END ELEMENT DEFINITIONS -->
</xsd:schema></types>

<!--BEGIN MESSAGE DEFINITIONS - DO NOT MODIFY THIS BLOCK!!! -->
<message name="deliverNotificationInputMessage">
    <part name="parameters" element="tns:deliverNotification"/>
</message>
<message name="deliverNotificationOutputMessage">
    <part name="parameters" element="tns:deliverNotificationResponse"/>
</message>
<message name="killChildsInputMessage">
    <part name="parameters" element="tns:killChilds"/>
</message>
<message name="killChildsOutputMessage">
    <part name="parameters" element="tns:killChildsResponse"/>
</message>
<message name="addListenerInputMessage">
    <part name="parameters" element="tns:addListener"/>
</message>
<message name="addListenerOutputMessage">
    <part name="parameters" element="tns:addListenerResponse"/>
</message>
<message name="setNodeTypeInputMessage">
    <part name="parameters" element="tns:setNodeType"/>
</message>
<message name="setNodeTypeOutputMessage">
    <part name="parameters" element="tns:setNodeTypeResponse"/>
</message>
```

```
<message name="getAlphaInputMessage">
    <part name="parameters" element="tns:getAlpha"/>
</message>
<message name="getAlphaOutputMessage">
    <part name="parameters" element="tns:getAlphaResponse"/>
</message>
<message name="getBetaInputMessage">
    <part name="parameters" element="tns:getBeta"/>
</message>
<message name="getBetaOutputMessage">
    <part name="parameters" element="tns:getBetaResponse"/>
</message>
<!--END MESSAGE DEFINITIONS -->

<gwsdl:portType name="ChessPortType"
    extends="ogsi:GridService ogsi:NotificationSource
        ogsi:NotificationSink">

    <!--BEGIN OPERATION DEFINITIONS - DO NOT MODIFY THIS BLOCK!!! -->
    <operation name="deliverNotification">
        <input message="tns:deliverNotificationInputMessage"/>
        <output message="tns:deliverNotificationOutputMessage"/>
        <fault name="Fault" message="ogsi:FaultMessage"/>
    </operation>
    <operation name="killChilds">
        <input message="tns:killChildsInputMessage"/>
        <output message="tns:killChildsOutputMessage"/>
        <fault name="Fault" message="ogsi:FaultMessage"/>
```

```
</operation>
<operation name="addListener">
    <input message="tns:addListenerInputMessage"/>
    <output message="tns:addListenerOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
</operation>
<operation name="setNodeType">
    <input message="tns:setNodeTypeInputMessage"/>
    <output message="tns:setNodeTypeOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
</operation>
<operation name="getAlpha">
    <input message="tns:getAlphaInputMessage"/>
    <output message="tns:getAlphaOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
</operation>
<operation name="getBeta">
    <input message="tns:getBetaInputMessage"/>
    <output message="tns:getBetaOutputMessage"/>
    <fault name="Fault" message="ogsi:FaultMessage"/>
</operation>
<!--END OPERATION DEFINITIONS -->

<!--BEGIN SERVICEDATA DEFINITIONS - DO NOT MODIFY THIS BLOCK!!! -->
<sd:serviceData name="ControlData"
    type="data:ControlDataType"
    minOccurs="1"
    maxOccurs="1"
```

```
            mutability="mutable"
            modifiable="true"
            nillable="false">
        </sd:serviceData>
        <sd:serviceData name="LocalData"
            type="data:LocalDataType"
            minOccurs="1"
            maxOccurs="1"
            mutability="mutable"
            modifiable="false"
            nillable="false">
        </sd:serviceData>
        <sd:serviceData name="PerfData"
            type="data:PerfDataType"
            minOccurs="1"
            maxOccurs="1"
            mutability="mutable"
            modifiable="false"
            nillable="false">
        </sd:serviceData>
        <!--END SERVICEDATA DEFINITIONS -->
    </gwsdl:portType>
    </definitions>
```