Optimización de la estrategía de Minimax utilizando HPC

Implementación para una variante del juego "Fox and Geese" en tiempo real

Pablo Andrés Grill Bermúdez
Montevideo, Uruguay
pablogrill21@gmail.com

Mathias Claassen Fast
Montevideo, Uruguay
mathiasclaassen7@gmail.com

***Resumen*— Minimax es una estrategia utilizada para la resolución de juegos de suma cero. Los juegos de suma cero se caracterizan por involucrar dos jugadores y la particularidad de que la ganancia de un jugador involucra una pérdida para el oponente. Minimax se basa en esta características de dichos juegos para tomar la decisión de que jugada es más conveniente realizar. Para ello, en cada instante de tiempo se decide que jugada realizar teniendo en cuenta que cada jugador se desempeñara de la mejor manera posible. Esto último se logra generando un árbol de ejecución donde cada nodo representa los distintos estados del juego. La estrategia de Minimax se implementa a menudo mediante un gran backtraking, lo que provoca que el tiempo de respuesta en ciertas ocasiones sea demasiado largo. El presente trabajo se enfoca en incluir paralelismo en la implementación de Minimax de manera de mejorar el tiempo de respuesta.**

Keywords—Minimax; Juegos de suma cero; paralelismo: tiempo de respuesta; backtraking

I. INTRODUCTION

En la asignatura de Programación Lógica del presente año uno de los obligatorios consistía en la implementación del algoritmo de Minimax para una variante juego "Fox and Geese". Dicha implementación se realizó en el lenguaje Prolog y permitía distintas variantes del juego. Una de ellas incluía la posibilidad de que uno de los jugadores sea la computadora. La emulación del comportamiento de la computadora se realizó mediante la estrategia de Minimax.

En la implementación desarrollada se incluía la posibilidad de variar el nivel de dificultad del juego. Esto se lograba incluyendo diferentes niveles de profundidad para la ejecución del algoritmo de Minimax. Al aumentar la profundidad del algoritmo se podía tener mayor información de cada posible movimiento lo que permitía tomar una mejor decisión.

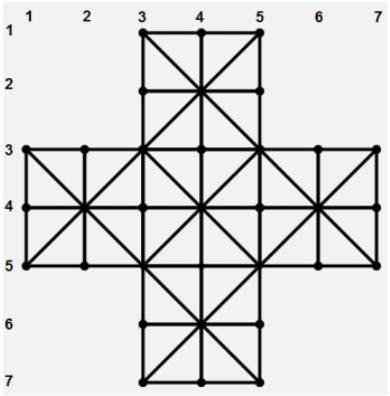
Uno de los inconvenientes de la solución planteada consistía en que al incrementar el nivel de profundidad el tiempo de respuesta crecía considerablemente. Este hecho se manifestaba ya que el árbol de ejecución crece de forma exponencial, requiriendo de un tiempo de cómputo muy grande. Esta particularidad provocaba que un juego contra el computador con grandes niveles de profundidad no sea dinámico.

Una alternativa, la considerada en este trabajo, consiste en emplear el procesamiento paralelo al momento de implementar el algoritmo de Minimax de manera que el tiempo de respuesta pueda disminuir de forma considerable, permitiendo de esta forma la posibilidad de mantener un juego dinámico con mayores niveles de profundidad.

II. DESCRIPCIÓN DEL PROBLEMA

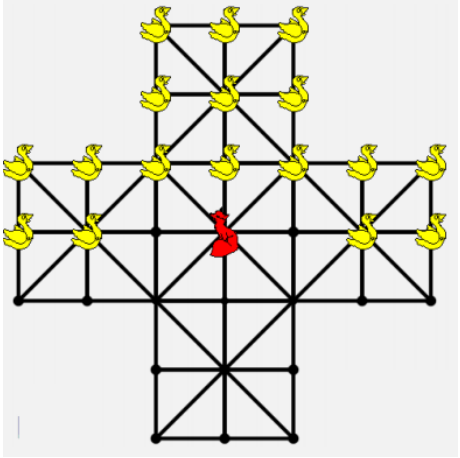
El problema a desarrollar consiste en la implementación de una versión paralela del algoritmo de Minimax para un juego de suma cero. Para este proyecto se optó por utilizar una variante del juego "Fox and Geese". En dicho juego participan 2 jugadores; uno de ellos se encarga de mover la ficha del zorro y el otro se encarga de mover los gansos.

En la versión utilizada se dispondrá de un tablero de 7x7, en el cual algunas de las posiciones no estarán disponibles.



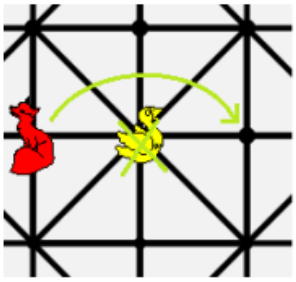
Los movimientos y objetivos del juego varían dependiendo del tipo de ficha que se está utilizando. El objetivo de los gansos consiste en dejar sin movimientos posibles al zorro. El objetivo del zorro es comer un cierto número de gansos. En particular, en esta versión se iniciará el tablero con 17 gansos y un zorro, ganando el zorro cuando coma un total de 12 gansos.

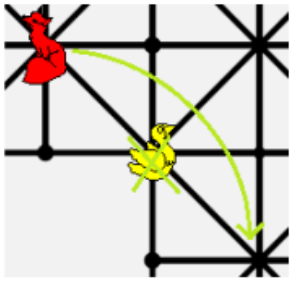
La disposición inicial de las fichas será la siguiente:



Los gansos tienen un solo tipo de movimiento, el cual consiste en moverse a cualquier celda adyacente que no esté ocupada. En el caso del zorro existen dos posibles movimientos. El primer movimiento es análogo al del ganso. El otro consiste en comer un ganso. Para realizar este último movimiento el ganso debe encontrarse en una celda adyacente al zorro, estando la celda posterior libre.

Dicho movimiento se detalla con mayor exactitud en la imagen.





Dadas las características del juego se pueda apreciar fácilmente que es un juego de "Suma Cero", por lo que la utilización de Minimax surge de forma intuitiva.

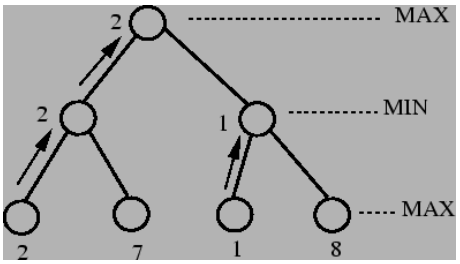
III. UTILIZACIÓN DE HPC

Como ya se mencionó anteriormente, las implementaciones de Minimax se desarrollan típicamente mediante un backtracking. Dicho backtraking genera un árbol en el cual cada uno de los nodos se corresponde con una configuración específica del tablero y los arcos representan una posible jugada. El árbol tiene por raíz la configuración actual del partido y los hijos se generan siguiendo posibles secuencias de jugadas de cada uno de los jugadores. Los nodos del árbol pueden ser terminales o internos. Los nodos terminales se alcanzan en dos ocasiones, o bien se alcanzó un escenario de fin de juego para alguno de los participantes, o bien se alcanzó la profundidad definida para el algoritmo.

El árbol generado se divide en dos tipos de niveles dependiendo del jugador que deba mover en ese momento. Dichos niveles se van alternando uno a uno emulando la jugada de cada jugador.

La generación de este árbol se realiza con el objetivo de poder tomar la decisión de que jugada realizar. Esta decisión se construye realizado diferentes evaluaciones en cada uno de los nodos. Los nodos más fáciles de evaluar son los terminales. La evaluación en dichos nodos consiste en aplicar una función de evaluación predefinida. En el caso del juego "Fox and Geese", como los objetivos de los jugadores dependen del tipo de ficha se deben definir 2 funciones de evaluación.

La evaluación en los demás nodos depende del nivel en que nos encontremos. En caso de ser un nivel del jugador actual se selecciona la rama hija que genere una mayor ganancia. En caso de ser un nodo correspondiente a una movida del contrincante se selecciona la rama hija que genere la menor ganancia. De esta forma podemos diferenciar los nodos del árbol en "nodos Min" y "nodos Max" de pendiendo del nivel en que nos encontremos.



Uno de los principales inconvenientes de la implementación del algoritmo de Minimax radica en que a medida que se van agregando niveles el árbol crece de manera exponencial. Esto provoca que el tiempo de cómputo necesario para tomar la decisión de que jugada realizar sea muy alto. La primer mejora que se puede realizar al backtaking es realizar algunas podas. En el algoritmo de Minimax se pueden realizar dos podas básicas, las denominadas podas alfa y podas beta. Para realizar estas podas es necesario mantener 2 valores en la ejecución, los denominados alfa y beta. El valor alfa consiste en el valor de la mejor jugada hasta el momento, el valor de beta consiste en la mejor jugada del oponente hasta el momento.

La poda alfa se realiza en los nodos Min y consiste en descartar la búsqueda debajo de cualquier nodo cuyo beta sea menor o igual al valor del alfa de cualquiera de sus ancestros. La poda beta se realiza en los nodos Max y consiste en descartar la búsqueda debajo de cualquier nodo cuyo alfa sea mayor o igual al valor beta de cualquiera de sus ancestros.

Si bien es verdad que las podas alpha y beta mejoraban considerablemente el tiempo de respuesta del algoritmo, en la implementación realizada con Prolog pudo apreciarse que al utilizar más de 5 niveles de profundidad el tiempo de respuesta era de varios minutos. Esto provocaba que mantener un juego dinámico contra la máquina no sea posible para niveles de profundidad superiores.

Esta limitante del algoritmo se produce debido a que el árbol de Minimax crece de forma considerable a medida que se agregan diferentes niveles. Para obtener la jugada a realizar es necesario procesar todas las ramas del árbol, lo que provoca que su ejecución en un programa secuencial crezca de manera considerable.

Ante este hecho surgió la idea de utilizar la programación paralela de manera de lograr obtener una mejor performance en el algoritmo y lograr la posibilidad de utilizar niveles más profundos que 5.

Sumado a esto, el algoritmo de Minimax tiene la particularidad de que el cálculo de 2 nodos pertenecientes al mismo nivel es independiente. Esta característica del algoritmo permite que el mismo sea altamente paralelizable. Utilizando varios procesos se pueden dividir las ramas del árbol de manera que se distribuya de manera equitativa el número de nodos a procesar. Esto permitirá que el tiempo de respuesta pueda mejorarse.

IV. ESTRATEGÍA DE PARALELIZACIÓN

Observando en detalle el algoritmo de Minimax se puede apreciar que el mayor tiempo de ejecución se encuentra en el procesamiento de los distintos nodos del árbol generado. Otro aspecto que se destaca del algoritmo, que ya se menciono anteriormente, es que el procesamiento de dos nodos es independiente. Esto sucede ya que los cálculos del valor de cada nodos solo dependen de sus nodos hijos o del propio nodo en caso de nodos terminales.

La estrategia de paralización diseñada consiste en dividir los nodos del árbol de manera que el cálculo del algoritmo se distribuya entre distintos procesos. De esta forma, al reducir el dominio de cálculo de cada proceso el tiempo de ejecución debería disminuir de forma considerable. Además, el hecho de que cálculos de nodos hermanos son independientes permite que la comunicación entre distintos procesos sea reducida, lo que permite explotar de mejor manera las unidades de procesamiento con mejor performance. Teniendo en cuenta los puntos destacados anteriormente se diseñó una solución al problema utilizando el lenguaje de programación C y la biblioteca MPI.

A. Función de Evaluación

Para poder implementar el algoritmo es necesario tener una función de evaluación para poder aplicar en los nodos terminales. Teniendo en cuenta los diferentes tipos de ficha es necesario definir 2 funciones, una para el zorro y otra para los gansos. Las funciones que se decidieron utilizar son las siguientes.

- zorro → 10*(17-cantg) + cantMovZorro + cantGansosVulnerables(Tablero);
- ganso → 7*(cantg) + 4*(8-cantMovZorro) + 10*ComponentesConexasGansos(Tablero);

Dichas fórmulas se basaron fuertemente en las desarrolladas en el Obligatorio de Prolog, las cuales habían demostrado comportarse de manera correcta. Además las formulas contemplan aspectos de ataque y defensa, lo que permite que el desempeño de ambos jugadores sea eficiente.

De esta forma, el cálculo de los nodos terminales puede realizarse de forma trivial con un único proceso. En los nodos intermedios la estrategia consistirá en dividir la carga de trabajo en varios procesos. De esta forma se asignarán algunos nodos hijos a varios procesos de manera de que el trabajo pueda paralelizarse.

B. Utilización de MPI-2

Teniendo en cuenta que la construcción del árbol se realiza de forma iterativa y dinámica es imposible conocer desde el comienzo del algoritmo la estructura final del árbol de ejecución. No conocer el dominio total al comienzo del algoritmo no permite realizar una distribución estática de manera equitativa. Teniendo este aspecto en cuenta se diseño el algoritmo considerando la asignación de tareas de la manera más dinámica posible. De esta manera se contemplaba logra modelar de mejor manera el dinamismo del algoritmo de minimax.

Analizando las diferentes alternativas que proveía MPI llegamos a la conclusión de que la variante de MPI-2 era la que más se ajustaba a nuestros requerimientos. El principal motivo de tomar la decisión de utilizar dicha versión de la biblioteca fue el hecho de que contenía la operación colectiva MPI_Comm_spawn. Dicha operación no se encuentra en el estándar básico de MPI y brinda la posibilidad de crear procesos de manera dinámica. Teniendo en cuenta las características dinámicas del algoritmo de Minimax, la posibilidad de realizar spawn de nuevos procesos permite obtener resultados más óptimos y eficientes.

C. Arquitectura de la Solución

Para el presente proyecto se realizó el desarrollo de 3 diferentes implementaciones del algoritmo para poder realizar comparaciones y extraer conclusiones. Por un lado se realizó la implementación de dos algoritmos seriales. El primero de ellos es una versión sin ninguna optimización del algoritmo de Minimax. La segunda versión del algoritmo si bien continúa siendo serial contiene algunas optimizaciones que permiten obtener resultados más performante. Las optimizaciones consisten en incluir en el algoritmo las podas alfa y beta mencionadas anteriormente. Por último se realizó una implementación del algoritmo mediante la utilización de MPI-2. Dicha implementación utiliza en gran medida el procesamiento en paralelo y las operaciones de conjunto que brinda MPI.

Continuaremos definiendo en grandes rasgos la estructura del algoritmo paralelo desarrollado. Para el mismo se crearon 3 programas diferentes (interface_MPI, minimax_nivel1, minimax_MPI). Describiremos brevemente cada uno de ellos

El programa principal es el interface_MPI. Dicho programa se encarga de procesar los eventos de entrada y salida de la interfaz de usuario. Además, en el caso en que se selecciona por la máquina juegue, se encarga de comenzar la paralelización del algoritmo de Minimax. La estrategia que realiza es la siguiente; dado el tablero y jugador actual el

programa calcula todos los posibles movimientos a realzar. Dicho valor dependerá mucho del jugador actual. En promedio el zorro entre 5 y 8 movimientos y los ganso entre unos 12 o 20. Luego el programa invoca mediante la operación MPI_Comm_Spawn la generación de procesos mininimax_nivel1 y asigna a cada uno de ellos un posible movimiento. Luego mediante la operación de MPI_Broadcast comunica a cada uno de los procesos hijos variables importantes del algoritmo como lo son la profundidad y el jugador actual. Posteriormente espera por la respuesta de cada uno de los hijos, escogiendo la que da mejor ganancia para realizar el movimiento.

El programa minimax_nivel1 se encarga de resolver el subárbol asignado y devolver el resultado de la ganancia al su proceso padre (interface_MPI). Para realizar este cálculo vuelve a utilizar el procesamiento paralelo. Para ellos invocara mediante la operación de MPI_Comm_Spawn la creación de n procesos del programa minimax_MPI. El valor de n dependerá del jugador actual. En caso que sea un zorro será 10 y en caso de los gansos serán 4. Una vez que dicho procesos están creados transmitirá a los mismos las variables recibidas y esperara a que el master de los procesos hijos comunique el movimiento de mayor ganancia. Una vez recibido dicho valor lo comunica al padre y finaliza. Este programa no realiza ningún cálculo, simplemente se encarga de generar los procesos hijo y comunicar el resultado del subárbol al padre.

Por último, el programa minimax_MPI es el encargado de resolver efectivamente el subárbol generado. Cada subárbol se resuelve con un conjunto de procesos, por ellos el programa trabaja de manera colectiva. Primeramente el proceso master (el de rango cero dentro de su comunicador) se encargara de calcular todos los posibles movimientos del tablero asignado. Una vez realizado ese cálculo completa con tableros "nulos" hasta tener un valor divisible por el número de procesos del grupo. Los tableros son asignados mediante la operación MPI_Sacatter. Cada uno de los procesos resuelve su conjunto de tableros aplicando el algoritmo de Minimax con poda. Los tableros nulos se ignoran. Los resultados de cada proceso se comunican mediante la operación MPI_Gather y el proceso master es el encargado de transmitir el resultado al padre(MPI_nivel1).

En la estructura anteriormente descripta claramente puede apreciarse que se está realizando una paraleización en 2 niveles al algoritmo de minimax básico.

D. Balance de carga

Teniendo en cuenta el dinamismo del algoritmo el balance de carga era una tarea crucial para este proyecto. Teniendo en cuenta que la estructura final del árbol de ejecución no es conocida de antemano la asignación estática estuvo descartada en todo momento. La asignación se realizó de manera de contemplar el dinamismo de la mejor manera posible.

Por un lado, al utilizar MPI-2 los procesos se generan a medida que son necesarios. Esto permite una mejor utilización de los recursos ya que los mismos se van liberando a medida que culminan sus tareas. Por otro lado, al momento de distribuir los tableros en el programa minimax_MPI la misma se realiza de manera equitativa con cada uno de los procesos. La distribución es equitativa ya que cada proceso recibe un mismo número de tableros. Además, los tableros nulos se dispersaron de manera equitativa entre los diferentes procesos. Con estas consideraciones la carga de cada proceso es similar.

Además, al utilizar MPI_Comm_Spawn el número de procesos finales dependerá altamente del escenario actual. Esto permite que en escenarios con muchos movimientos posibles se utilice en mayor número de procesos y sea el árbol de ejecución distribuido de mejor manera.

E. Tolerancia a Fallos

MPI no ofrece un mecanismo directo de tolerancia a fallos. Por ello en caso de que ocurra un fallo en cualquiera de los procesos generados MPI abortaría el total de la ejecución. Para evitar esto se agregaron algunos controles. Al analizar las ejecuciones del algoritmo vimos que el momento en que podía llegar a abortarse un programa era al procesar el subárbol mediante minimax.

Para evitar que dichos errores aborten la totalidad del programa se optó por incluir el manejo de errores en cada llamada a una operación colectiva de MPI. De esta manera podía procesarse el error y evitar que aborte la totalidad del algoritmo. Teniendo en cuenta que en el algoritmo de Minimax es utilizado para la resolución de juegos es más importante el tiempo de respuesta del algoritmo que la exactitud de la solución. Con esa premisa en consideración optamos por ignorar los tableros que generaban errores y solo considerar los tableros correctos. Si bien esta estrategia no aseguraba obtener el mejor movimiento permitía obtener un juego más fluido que en caso de que se re-procesara todo el tablero. Además, considerando que la respuesta se calculaba en función de los procesos restantes el comportamiento del algoritmo continuaba siendo correcto.

Al realizar pruebas al respecto pudimos apreciar que los casos en que se producía algún error eran mínimos, por lo tanto la solución planteada generaría el mejor movimiento en la mayor cantidad de los casos.

V. EVALUACIÓN ESPERIMENTAL

Para la evaluación experimental se calcularon los valores de speed up y eficiencia de las distintas versiones del algoritmo. Para poder realizar un análisis más profundo se realizaron diferentes pruebas variando la profundidad con la que se ejecutaba el algoritmo de minimax. Sumado a esto, en los casos de que el algoritmo utilizaba procesamiento paralelo se realizaron pruebas con distintos números de procesadores. Esto último se realizó utilizando un archivo de hostfile al momento de invocar la ejecución del programa.

Considerando que el algoritmo presentaba un gran dinamismo se opto por realizar las pruebas ejecutando los programas sin jugadores humanos. De esta forma se dejaba ejecutar cuatro movimientos del algoritmo y se medía el tiempo. En los cuatro movimientos se incluían dos ejecuciones de minimax con jugador actual zorro y dos con ganso. Además se utilizó un tablero diferente al inicial que garantizaba un gran

número de movimientos posibles para cada uno de los jugadores.

Se realizaron varias ejecuciones por cada implementación y se registro el tiempo promedio de cada una de ellas. En los casos de los niveles impares se opto por registrar el tiempo diferenciando por jugador. Se tomó esta decisión considerando que el costo de mover los gansos es mayor ya que en promedio el número de movimientos posibles es mayor (17 a 5 aproximadamente) .

Se registraron los tiempos a partir del nivel 7 ya que en niveles anteriores el tiempo era menor a 1 segundo.

Las pruebas se realizaron en las salas de máquinas de la facultad, ejecutando la versión de MPI ubicada en /ens/home01/hpc00/bin/mpich2.1.5. En el caso paralelo se decidió incluir en el hostfile 2 procesadores por máquina.

De esta manera se obtuvieron los siguientes tiempos de ejecución.

	Alg. Serial	*Alg. serial con poda*	*Alg. Paralelo MPI 32 proc*	*Alg. Paralelo MPI 42 proc*	*lg. Paralelo MPI 54 proc*
Prof 7	Zorro=10s Ganso=30s	<1s			
Prof 8	1 m,30s	~2s			
Prof 9	>11 m	Zorro=8s Ganso=20s	Zorro=8s Ganso=20s		
Prof 10	N/A	1min,20s	30s	20s	20s
Prof 11	N/A	Zorro=3m Ganso=6m	Zorro=1m Ganso=3m	Zorro=1m Ganso=2 m	Zorro=1m Ganso=2m
Prof 12	N/A	22m	10m	10m	5m

A continuación procederemos con el cálculo del Speed up algorítmico. Considerando que el algoritmo con poda fue el mejor algoritmo serial que pudimos implementar será tomado como el valor de T1 para los diferentes casos. Se calculará el valor de Speed up para cada uno de los niveles de profundidad considerados. Además en los casos en que se diferenciaron los tiempos de los zorros y gansos se calculara por separado

Profundidad 10

S32=80s/30s=2.66

S42=80s/20s=4

S54=80s/20s=4

Profundidad 11

Zorro

S32=180s/60s=3

S42=180s/60s=3

S54=180s/60s=3

Ganso

S32=360s/180s=2

S42=360s/120s=3

S54=360s/120s=3

Profundidad 12

S32=1320s/600s=2.2

S42=1320s/600s=2.2

S54=1320s/300s=4.4

Ahora procederemos a calcular la eficiencia computacional del algoritmo para los casos anteriores.

Profundidad 10

E32=2.66/32=0.08

E42=4/42=0.09

E54=4/54=0.07

Profundidad 11

Zorro

E32=3/32=0.09

E42=3/42=0.07

E54=3/54=0.05

Ganso

E32=2/32=0.07

E42=3/42=0.07

E54=3/54=0.06

Profundidad 12

E32=2.2/32=0.07

E42=2.2/42=0.05

E54=4.4/54=0.08

VI. CONCLUSIONES

A continuación comentaremos las conclusiones que pudieron extraerse del desarrollo del presente proyecto. Comenzaremos analizando el algoritmo serial sin ninguna optimización. Claramente se puede apreciar la exponencialidad con la que crece el algoritmo de Minimax. Variar la profundidad de 8 a 9 produce que el tiempo se incremente en un factor de 11. Esto ocurre debido a que el número de movimientos posibles se incrementa lo que genera que el árbol de ejecución sea mayor. Como el algoritmo serial básico no aplica ninguna "poda", para obtener el movimiento a realizar el árbol de ejecución debe ser recorrido y analizado en si totalidad.

Desde el punto de vista de la usabilidad del algoritmo básico, teniendo en cuenta que la idea es utilizar la estrategia de Minimax para lograr un juego fluido y dinámico contra los humanos, dicho algoritmo se encuentra correcto para nivel 6 y tolerable para niveles 7 y 8 . Sin embargo, para valores de profundidad superiores el algoritmo deja de ser viable ya que ninguna persona estaría dispuesta a esperar un tiempo superior a 10 minutos.

Continuaremos analizando el algoritmo una vez que se incluyeron en el mismo las podas alfa y beta. La mejora de performance que se obtuvo al aplicar dicha mejora fue muy importante. En niveles inferiores a 8 el tiempo de respuesta es casi instantáneo. En nivel de profundidad 9 el contraste en el algoritmo básico y el que contiene poda es evidente. El tiempo de respuesta del algoritmo básico es aproximadamente 60 veces mayor.

Esta diferencia sustancial de performance se produce debido a que en los casos que se aplica la poda se reduce el árbol de ejecución del algoritmo. El podar ramas que de ante mano se sabe que no generaran ningún resultado productivo permite no desperdiciar los recursos de cómputo. Considerando además el aspecto de que las podas alfa y beta solo descartan ramas no productivas, el resultado obtenido continua siendo el optimo; lo cual eleva el valor ganado con la aplicación de las podas.

Por otro lado debe considerarse además que las podas alfa y beta son mejoras a nivel del diseño del algoritmo. Para utilizar esta versión no es necesario de disponer de una gran infraestructura de hardware ya que aún continua siendo una implementación serial. En esta implementación del algoritmo la mejora de performance no se logra a causa de un paralelismo ni de una distribución del trabajo sino que se logra debido a una re-estructuración y mejor diseño del algoritmo.

Con respecto a la usabilidad del algoritmo, podemos ver que el mismo permite mantener un juego dinámico hasta el nivel 9 y un juego tolerable hasta el nivel 10. Estos valores son 2 niveles más altos que los del algoritmo serial básico.

Ahora destacaremos algunas conclusiones en lo que respecta a la implementación del algoritmo de manera paralela utilizando MPI. Considerando que dicha implementación se basa en el algoritmo con poda realizaremos la comparación en contraste con el mismo. Al analizar los diferentes niveles de profundidad podemos apreciar que el algoritmo que utiliza el procesamiento en paralelo obtiene una mejor performance.

En el caso de el nivel 9 los tiempos son similares. Esto se debe a que el algoritmo serial es muy eficiente en esa profundidad. Al utilizar el procesamiento en paralelo surgen costos de comunicación que no están contemplados en el caso serial. Por lo tanto, con tiempos tan pequeños la ganancia del algoritmo casi que no se aprecia. En el caso de profundidad 10 ya se puede apreciar una mejoras importante en el tiempo de respuesta del algoritmo. En esta ocasión el tiempo del algoritmo serial es 4 veces mayor que el del algoritmo paralelo. En este nivel en el caso paralelo el juego es dinámico y en el caso serial es tolerable.

En el caso de profundidad 11 la ganancia es similar que en el de profundidad 10. El algoritmo paralelo se comporto entre 3 y 4 veces más perforamante que el algoritmo serial.

El caso de profundidad 12 será analizado de manera independiente de los demás ya que presenta características diferentes. En particular en este caso el rango de datos a procesar es mucho mayor que en los casos anteriores. La exponencialdad del algoritmo en lo que respecta al crecimiento produce que por más que se pode de manera eficiente el tiempo de ejecución sea superior a los 20 minutos. Además, estamos ante un caso en que el mejor tiempo de ejecución del algoritmo paralelo fue de 5 minutos. Este valor no es ni siquiera tolerable para un juego contra un humano. Por lo tanto, el nivel de profundidad 12 es una cota superior. Independiente de ese hecho, en dicho nivel de profundidad es cuando se puede apreciar claramente la ganancia del procesamiento en paralelo. Al obtener valores tan altos en los tiempos de ejecución seriales, los costos de comunicación del algoritmo paralelo van teniendo un valor menor en comparación del tiempo de cómputo. Es esta situación en la que el utilizar el procesamiento paralelo comienza a tener sentido y se vuelve una alternativa viable. En tiempos de ejecución chicos (niveles menores a 11) la ganancia de performance del procesamiento paralelo se opaca a causa de los costos de comunicación.

Con respecto a la eficiencia computacional, considerando lo anteriormente mencionado podemos ver que la misma no fue muy buena a causa de que los tiempos del algoritmo serial resultaron muy buenos. En la implementación en Prolog (de la que surgió la del presente proyecto), en niveles 4 y 5 el tiempo ya se disparaba a decenas de minutos. Sin embargo, en los algoritmos en C utilizando de buena manera la memoria dinámica pueden obtenerse programas más performantes. Por otro lado, para niveles superiores a 12 comienza a verse viable la utilización del procesamiento paralelo ya que la eficiencia comienza a ser mayor debido a que los tiempos de comunicación son ínfimos en comparación con el tiempo de cómputo.

La viabilidad del algoritmo paralelo puede ser analizada desde dos puntos de vista diferentes. Desde el punto de vista funcional, utilizar el procesamiento paralelo para el presente

juego no es viable debido a que el costo de hardware necesario no fundamenta la ganancia. Esto es debido a que funcionalmente el algoritmo no debería utilizar niveles mayores al 10 y 11 para que sea dinámico y amigable al usuario. Sin embargo, desde un punto de vista teórico, al analizar niveles mayores a 12 el paralelismo comienza a ser productivo ya que los tiempo permiten que sea mejor explotado. Si bien es verdad que dicho niveles no tendrían una usabilidad práctica, parece oportuno destacar que en dichos casos el uso del algoritmo paralelo es fundamentado ya que el tiempo de respuesta del mismo es considerablemente menor.

Con respecto a la escalabilidad del algoritmo, el mismo se ve acotado a causa de algunas características de Minimax. Por un lado está el hecho de que si bien los cálculos de las ramas son independientes, en algunos momentos es necesario considerar todos los sub-arboles hijos. En estos puntos de sincronización de procesos, el algoritmo se ve limitado a esperar al proceso hijo que demore más. Si bien con la distribución equitativa entre procesos se mitiga de buena manera este riesgo, como los sub-árboles son desconocidos a primera instancia es imposible determinar que los mismos tengan costos similares. Por otro lado, las unidades de cómputo con las que realizamos las pruebas son compartidas, esto también puede provocar que un procesador presente una performance menor.

Por otro lado, al paralizar en 2 niveles,(como fue nuestro caso), se pierde un nivel de poda. Esto ocasiona que el algoritmo paralelo procese más datos que el serial con poda. En caso de utilizar más niveles de paralelismo tal vez se logren procesamientos más eficientes, sin embargo se tendría que investigar con mayor profundidad si la ganancia obtenida compensa perder niveles de poda.

Analizando la decisión de utilizar MPI para el desarrollo del proyecto podemos apreciar que la misma fue correcta. Las operaciones colectivas de MPI permitieron que el procesamiento paralelo se realizara de manera intuitiva y eficiente. En especial, las operaciones scather y gather permitieron distribuir y recolectar de manera equitativa y manteniendo un código limpio y entendible. Además, la flexibilidad con la que MPI permite crear tipos de datos derivados nos permitió comunicar "tableros" y "movimientos " de manera sencilla y transparente. Por otro lado, creemos que la utilización de MPI-2 fue correcta. El dinamismo del algoritmo no permitía conocer de antemano el total de procesos necesarios a utilizar. En caso de utilizar MPI sin spawn tendríamos que haber mantenido un poll de procesos lo cual habría complicado el algoritmo de manera sustancial

En líneas generales el proyecto demostró que si bien utilizar el procesamiento paralelo permite mejorar la performance del algoritmo de manera significativa no es viable su utilización ya que los costos computacionales no fundamentan la ganancia de tiempo y niveles. Por otro lado, para investigaciones más teóricas con niveles de profundidad más altos el algoritmo paralelo se vuelve útil ya que la ganancia de tiempo se vuelve más importante y la eficiencia computacional aumenta. Por último nos parece oportuno destacar el hecho de como puede aumentarse la performance con solo mejorar el algoritmo. Las comparaciones entre el minimax básico y con poda son abismales y simplemente se realizaron cambios y mejoras en el algoritmo.

VII. REFERENCIAS

- http://es.wikipedia.org/wiki/Minimax#Optimizaci.C3.B3n
- http://es.wikipedia.org/wiki/Poda_alfa-beta
- http://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/node98.htm
- http://www.open-mpi.org/doc/
- http://beige.ucs.indiana.edu/I590/node85.html
- http://static.msi.umn.edu/tutorial/scicomp/general/MPI/content6.html