Universitat
Autònoma
de Barcelona

ULTRA DEEP BLUE:
THE ULTIMATE CHESS PLAYER

Memòria del Projecte Fi de Carrera
d'Enginyeria en Informàtica
realitzat per
Jaume Devesa Gómez
i dirigit per
Miquel Àngel Senar
Bellaterra, 17 de Setembre de 2008

Index

1.Introducció

1.1 Plantejament i objectius

Encara que en l'actualitat els ordinadors són percebuts com maquines amb desenes de funcionalitats, des del clàssic editor de textos a complicades aplicacions multimèdia, passant per programes client/servidor dins de l'univers d'Internet o elements d'oci amb milers de jocs a la nostra disposició, no hem d'oblidar que encara ens serveixen per el que van ser dissenyats i desenvolupats: com a avançades màquines de càlcul.

En els últims 50 anys, disciplines com la criptografia han hagut de evolucionar a mesura que els ordinadors han sigut mes potents i han resolt amb pocs minuts càlculs que abans es consideraven massa complexos.

Però hi ha un altre tipus de problemes on la complexitat del problema supera encara, avui en dia, la capacitat de càlcul de qualsevol super ordinador. Aplicacions de predicció del temps, de descodificació el genoma humà, estudi de les senyals del projecte SETI... han fet sorgir un nou paradigma de càlcul basat en la col·laboració de diferents ordinadors en un càlcul distribuït.

Aquest projecte intenta, en molt petita escala si tenim en compte els exemples anteriors i en un exemple molt concret, fer un estudi sobre els avantatges que ens proporciona el càlcul distribuït, així com els desavantatges de no centralitzar el treball en una màquina com el retard en temps (*delay*) que suposa enviar informació a traves de una xarxa, o els problemes derivats de la falta d'informació sobre les variables que tracten les altres execucions que s'executen al mateix temps.

A grans trets, el programa presentat es una aplicació *GNU Chess* amb un motor de cerca basat en una evolució de l'algorisme del *MinMax* amb poda *alpha-beta*, adaptat perquè la millor opció de la següent jugada es calculi al mateix temps en diferents ordinadors de la ETSE a través del sistema de cues *Condor.*

És interessant veure l'evolució del algorisme de cerca, desenvolupat a partir d'una mentalitat purament seqüencial (i molt òptima: en cap moment és objectiu del projecte millorar la recerca pels nodes ni l'heurística que aplica), a una execució totalment paral·lela, i finalment, a partir d'un estudi previ, intentar arribar a un compromís entre els dos paradigmes per a aprofitar el millor de cada una de les execucions.

1.2 Planificació del treball que s'ha realitzat.

La majoria de projectes de final de carrera, poden tenir una part d'exposició del problema, una part d'evolució de costos, una part de l'estudi de la viabilitat, i una part de desenvolupament que tindrà un pes especific més gran en l'avaluació.

Segons aquests paràmetres, creiem que aquest no ha sigut un projecte que segueixi aquestes pautes. Al focalitzar un problema tant concret i tenint en compte el que volíem utilitzar però sense saber on arribaríem, la part d'estudi i la part de desenvolupament acaparen, segons el meu punt de vista, a parts iguals la importància dins del projecte, encara que és a la segona part a la que li hem dedicat més esforços.

La feina que es va portar a terme en el desenvolupament del projecte es va distribuir més o menys de la següent manera, intentant seguir la pauta entregada a mitjans de febrer com a informe de seguiment:

Durant els primers dos mesos, es va posar molta atenció en l'estudi del algorisme, i

la seva estructura en forma de fitxers '*.c'. L'objectiu era identificar l'algorisme de cerca de moviments i escollir el lloc idoni per 'trencar-lo', i a partir d'allà, gestionar la nostra evolució.

Llavors es tractava de crear un binari a part que executés l'algorisme de cerca remotament (*ParalelSearch*), i crear una execució paral·lela dels dos binaris, comprovant que la seva execució i el seu retorn no alteraven el funcionament normal del programa. Donat el gran volum de dades que es necessita per executar l'algorisme, ja que hi entren variables globals, funcions heurístiques, estructures de dades que mantenen l'estat de la partida, etc... aquesta va ser una evolució molt més costosa del que esperàvem.

Cap al 4rt mes de projecte, vaig centrar esforços en la implantació de *Condor* a la maquina pròpia; barallant-me amb la seva configuració. Un programa '*Hello World*' em va ajudar a fer les proves. El següent pas va ser automatitzar la crida de *Condor* des del programa *GNU Chess*, i fent referència al *ParalelSearch* per tal d'integrar els dos sistemes.

A partir d'aquí es va desenvolupar la part 'forta' del projecte, dividint la feina per la execució paral·lela. Llavors es va dedicar un temps al anàlisi teòric de l'aplicació realitzada, i, aplicant els coneixements adquirits a la primera part i les petites proves d'execució de la segona, es va dedicar un temps per a arribar a una execució mixta, un algorisme que es va proposar per a millorar l'anterior aprofitant les avantatges del algorisme original. Cosa que millor deixi per altres apartats posteriors.

La resta del temps, que abasta els dos últims mesos d'estiu i la primera setmana de setembre, s'ha dedicat a la redacció de la memòria i a les proves, així com a preparar el *PowerPoint* per a la defensa del projecte davant del tribunal.

Finalment, puc afirmar que els terminis assignats a cada part del desenvolupament de la feina van estar correctament marcats, ja que han permès evolucionar cada part amb un període de temps suficient per abastar els possibles errors que podien anar sorgint.

La única excepció, si més no, s'ha donat en la part de les proves finals. Donat el canvi d'ordinadors en les aules de la ETSE no es van poder portar a terme fins a principis de setembre. Això i uns errors no contemplats a última hora (de codi, i de sobrecarrega de la màquina *aocegrid*) ens han fet arribar una mica justos a l'entrega final.

1.3 Estructura de la Memòria

Aquesta memòria està estructurada de la següent manera:

L'apartat 1 en el que ens trobem és la introducció, on es posen les bases del projecte i es marca, de forma generalitzada, l'objectiu del projecte i la cronologia de la seva evolució.

L'apartat 2 es fixa sobretot en els detalls tècnics del *GNU Chess* i del *Condor*, per exposar les tecnologies en les que ens estem barallant.

L'apartat 3 mostra el desenvolupament del projecte en totes les seves fases, visitant aspectes tècnics del codi original i comparant-los amb la nostra evolució. Intentaré, en la mesura que em sigui possible, mostrar-ho a través d'esquemes i evitant l'aparició del codi. Crec que és una mesura que facilita la comprensió.

Donats una sèrie d'inconvenients de l'aplicació desenvolupada al apartat 3, l'apartat 4 proposa un nou algorisme per a obtenir una execució més òptima. De la mateixa manera, exposaré tant gràficament com em sigui possible la evolució portada a terme.

He destinat l'apartat 5 per al resultat de les proves i les conclusions, mostrant un banc de proves que es repeteix per als tres algorismes proposats, posant en

evidencia els avantatges i carències de cadascun d'ells.

Un epíleg de conclusions menys tècniques i més 'filosòfiques' (si se'm permet), tancarà la part explicativa de la memòria.

Després de tot, adjuntarem una sèrie d'annexes que mostraran fitxers de configuració i taules, que podran ser consultats per contrastar els valors esmentats a les proves.

No he considerat necessari fer una bibliografia. El tema és tan concret en alguns casos, que la informació es pràcticament inexistent (creieu-me si dic que no he trobat un sol lloc on especifiqui massa l'algorisme de *GNU Chess*, les explicacions són totes obtingudes consultant el codi); i en altres casos són massa generals (trobo absurd posar, per exemple, un manual de C o del mateix *Condor*; és molt fàcil aconseguir aquesta informació a través de Internet).

El que sí he inclòs, i com a lector agraeixo sempre, són petites explicacions o referències a pagines web al peu de pàgina. A mode de extensió de coneixements i inclús com a forma estètica, sempre m'han agradat molt i ara que tinc la ocasió, faig ús d'ells. El fet d'estar a la mateixa pàgina del contingut i no en una referència al final del informe crec que millora la utilitat d'una bibliografia.

2. Anàlisi dels recursos i tecnologies.

Ja hem apuntat a la introducció que l'objectiu d'aquest projecte es la integració del *GNU Chess* amb el *Condor*, i el seu posterior estudi del rendiment que dóna. De moment, però, aquest pretén ser un apartat que reculli els conceptes que ens situaran en el punt de partida del projecte i que corresponen a l'estudi previ abans de començar a desenvolupar. Per així conèixer amb què ens enfrontem i anar aclarint on volem arribar.

Els següents sub-apartats parlen de forma bastant exhaustiva de les característiques dels programes utilitzats, centrant-nos principalment en les parts que ens interessen: En el cas del *GNU Chess* no es la meva intenció fer un assaig sobre el programa, abastant tots els seus apartats, o proposar una heurística millor alhora de valorar les jugades, però si que intentaré explicar amb detall tot el que fa referència al seu algorisme de cerca. De la mateixa manera, no valoraré els possibles entorns de execució (universos) del *Condor*, però si que faré una parada en les comandes que m'han sigut útils alhora de configurar i desenvolupar i que es poden trobar en el codi font del programa *GNU Chess* que he adaptat.

2.1 Anàlisi del Sistema *Condor.*

2.1.1 Què es Condor?

Condor es un programa desenvolupat per el *Condor*® *Project* de la Universitat de

Wisconsin[1] que gestiona un sistema de cues d'execucions per a ordinadors en xarxa. En aquest projecte treballem amb la versió 6.8 de *Condor*. Tot i que no es *software* lliure, es pot descarregar gratuïtament a l'adreça marcada a (1).

El projecte Oliba de la UAB[2] es l'aplicació del sistema *Condor* en les màquines de la ETSE. M'he aprofitat del servei que ofereix el projecte Oliba a les màquines de la facultat per a poder desenvolupar el treball i així provar-lo en un entorn 'real'.

2.1.2 Com treballa Condor.

Condor treballa de manera que un Gestor Central s'encarrega de gestionar una sèrie d'execucions i de distribuir-les en una colla de màquines treballadores que es troben a la mateixa xarxa, de recollir els resultats, i de enviar-los als clients. *Condor* treballa amb una sola cua per gestionar totes les peticions, a diferència d'altres gestors de cues.

Una execució de *Condor* es basa en tres rols:

- Màquines client: Aquestes màquines són les que envien treballs per a ser executats en xarxa i reben el resultat d'aquestes operacions i els *logs* que generen. Des d'una màquina client pots comprovar com està en cada moment d'ocupat el sistema, gestionar l'execució dels teus treballs o establir característiques a les execucions (pots escollir la prioritat de l'execució i fins i tot diferenciar l'execució del treball segons el sistema operatiu).
- Màquines treballadores: Aquestes són les màquines que executen els treballs. La seva forma de treballar depèn de la configuració del fitxer de configuració local. Es pot establir una màquina treballadora dedicada a *full-time*, o aprofitar els moments que la màquina està en repòs (quan no hi han sessions obertes, o de nit) per a executar els treballs que el Gestor Central li envia.
- Gestor Central: El Gestor Central (*oliba.uab.es* en el nostre cas) és una màquina

1 *Http://www.cs.wisc.edu/condor*
2 *Http://www.oliba.uab.es*

dedicada a temps complert a gestionar les peticions de les màquines client i enviar-les a les màquines treballadores.

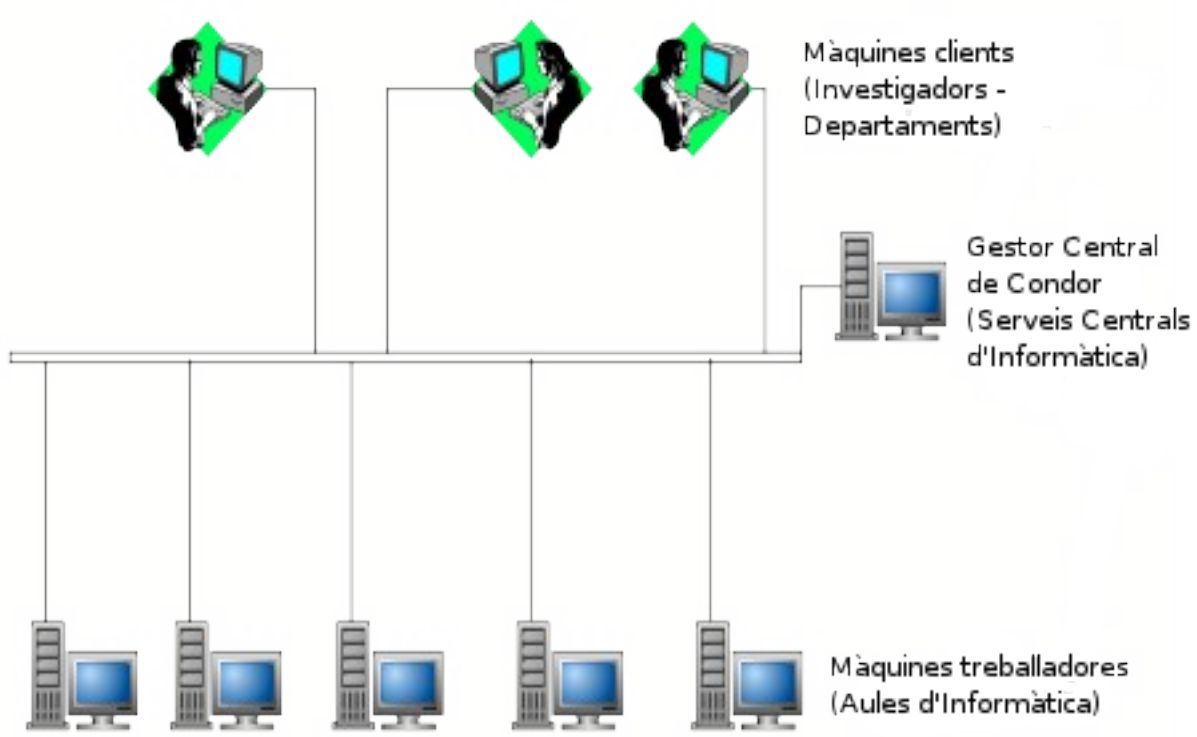


Figura 1: *Esquema bàsic del funcionament de Condor a la ETSE*

Condor no disposa d'una instal·lació diferent per a cada rol del sistema, sinó que el mateix programa està instal·lat totes les màquines. Un sistema de *deamons* es el que gestiona les tasques a fer per a cada una de les maquines. Els *deamons* que declarem que s'han d'aixecar quan s'engegui el programa els definim en el fitxer *condor_config.local* i serà el que determinarà el rol de la màquina dins del sistema.

Fem un repàs dels més importants:

- condor-master: aquest el el *deamon* central que s'encarregarà de engegar a tots els altres. Ha d'estar present en totes les configuracions.
- condor-startd: aquest deamon és el que dóna el rol de màquina treballadora. Si una màquina té aquest dimoni declarat, ens hem d'encarregar de dir-li en quins moments volem que treballi com a tal (a temps complert o només quan la tenim lliure) ja que aquest *deamon* necessita aquesta informació per gestionar les

execucions.

- condor-scheed: l'altre cara de la moneda. Aquest es el *deamon* encarregat de llançar treballs a executar. Si volem que la nostra sigui una màquina client, necessitem declarar aquest *deamon* al *condor-config.local.*
- condor-collector: el *condor-collector* recull la informació de la cua d'execució. Els altres *deamons* li envien informació periòdicament sobre l'estat de les màquines i les execucions dels treballs i aquest les recull i les gestiona. Amb la comanda *condor-status* ens podem informar de les dades del *collector* en qualsevol moment. És imprescindible doncs, si volem instal·lar un Gestor Central.
- Condor-negociator: aquest *deamon* agafa les dades del *collector, i startd* per tal de gestionar les peticions que li arriben de les maquines client (*scheed).* Qualsevol opció de prioritat que s'hagi definit alhora d'executar un treball es gestionarà des d'aquí. També es part fonamental del Gestor Central.

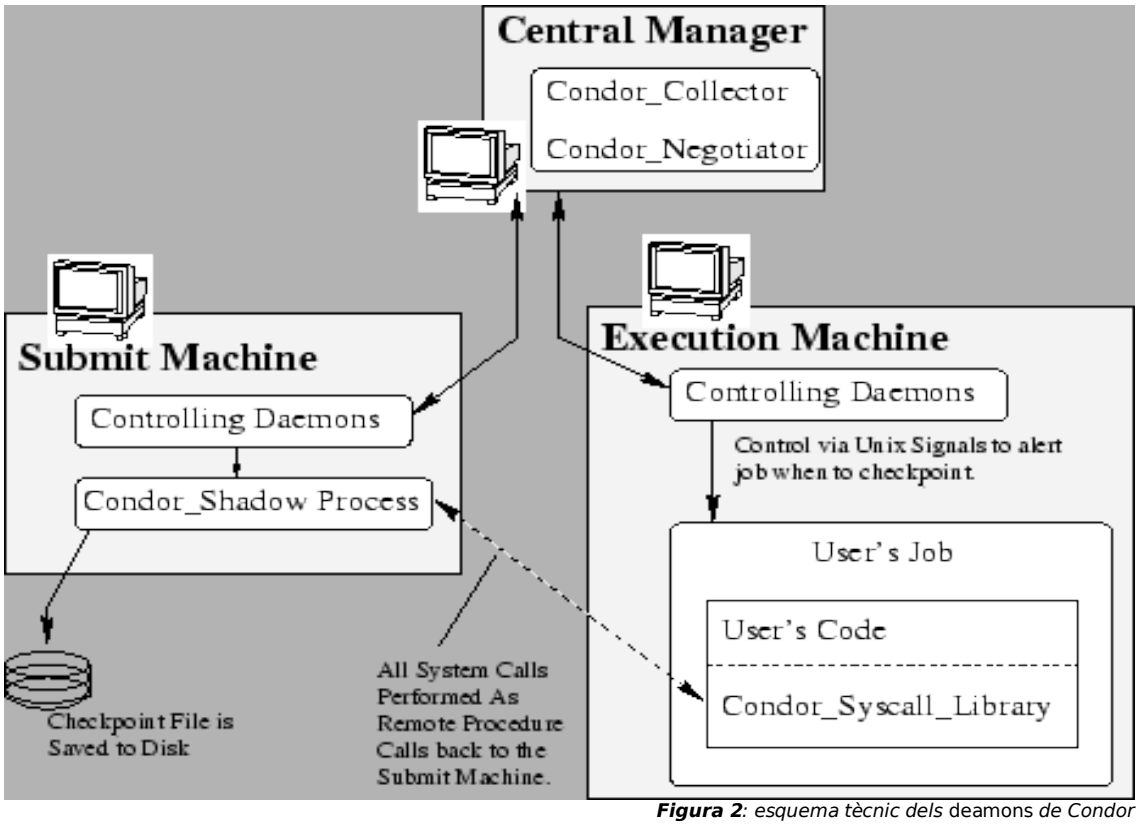


***Figura 2**: esquema tècnic dels* deamons *de Condor*

Per tant, l'us que li vulguem donar a *Condor* dependrà dels *deamons* que engeguem. Podem veure doncs, que tant podem tenir una màquina especialitzada en cada cas

com declarar més d'un rol per a poder aprofitar-la millor; o tal i com he hagut de fer jo alhora de desenvolupar, que una sola màquina sigui Gestor Central, treballadora i client.

Per poder-ho veure millor, he deixat en els annexos una còpia del fitxer *condor-config.local* tant en l'entorn de desenvolupament de la meva màquina com quan he executat les proves a la ETSE.

2.1.3 Execució de programes a Condor.

El que hem de fer per a enviar un treball a *Condor* és primer crear un fitxer de propietats on hi emmarcarem les característiques de l'execució i llavors enviar aquest fitxer com a paràmetre de la comanda *condor_submit*. Com sempre, no enumerarem totes les possibles propietats que pot tenir un fitxer de *Condor*, per això es pot consultar al manual d'usuari de la web[3] , sinó els que ens han servit per al desenvolupament del projecte. Ho explicarem amb un exemple:

```
# Inici del fitxer /home/jdevesa/files/file_example
Universe = vanilla
Executable = /home/jdevesa/bin/ParalelSearch
Output = condor_submit_out_$(Process)
Error = condor_submit_err
Log = condor_submit_log
Arguments =  3 72 37 96 37 2 $(Process)
Transfer_input_files = chess.param_$(Process)
Transfer_output_files = search.param_$(Process)
InitialDir = /home/jdevesa/files/
Should_transfer_files = YES
When_to_transfer_output = ON_EXIT
```

3*http://www.cs.wisc.edu/condor/manual/v6.8/*

```
Queue 30

# Final del fitxer /home/jdevesa/files/file_example
```

Aquest és un dels arxius de *submit* més complexos que he creat durant el desenvolupament del projecte. No podem dir que llançar una execució a *Condor* sigui quelcom difícil... Llistaré les propietats i les seves utilitats.

- Universe: Segons l'entorn de la xarxa i les propietats que volem d'execució, podem establir l'univers en el qual s'executaran els treballs. Per desgràcia, no he indagat gaire en les possibilitats d'aquesta opció, que crec, pot oferir grans rendiments amb la seva configuració. Per defecte, s'escull l'univers *vanilla.*
- Executable: Aquí li donem la ruta del programa que volem que s'executi remotament.
- Output, Error, Log: Són els fitxers de *output* (atenció, no es el fitxer que retorna, sinó la sortida estandard del programa, es a dir, allà on van a parar les comandes *printf*!), informe d'errors i de *logs* respectivament que crea *Condor* un cop acabada l'execució.
- Arguments: Els arguments d'entrada que necessita el programa executable.
- Transfer-input-files, Transfer-ouput-files: Els fitxer d'entrada i sortida de paràmetres del fitxer executable. Aquests, si no indiquem res en concret, els trobarem en mateix directori on es troba el fitxer binari que enviem a *Condor*.
- InitialDir: Si, en canvi, li volem donar una ruta on deixar els fitxers, li hem de dir en aquesta comanda.
- Should-transfer-files: Aquesta comanda pot estar a YES, IF_NEEDED i NO. Correspon a la necessitat d'enviar els fitxers esmentats anteriorment.
- When-to-transfer-output: Indica quan hem d'enviar el fitxer de sortida. ON_EXIT correspon al finalitzar l'execució, ON_EXIT_OR_EVICT envia el fitxer de sortida inclús quan es mata el procés.
- Queue: Indica el nombre de execucions que enviem a la cua de *Condor*. Va lligada amb la variable *$(Process)*. Aquesta variable es incremental per a cada execució. Per tant, podem veure que hem enviat 30 treballs junts a *Condor*, però tant els

arguments, com el fitxer *output*, com els fitxers d'entrada i sortida son diferents per a cada execució.

Llavors, escrivint *condor_submit /home/jdevesa/files/file_ëxample* a la línia de comandes, enviarem aquesta petició d'execució a *Condor.*

2.1.4 Altres comandes Condor

A part del *condor_submit* hem necessitat de més comandes de *Condor* per a portar a terme el projecte i que la seva enumeració servirà per comprendre millor la part on explico el desenvolupament del projecte. Són les següents:

- ./condor-master: aquesta comanda serveix per engegar el *deamon condor_master* i en conseqüència la resta de *deamons* que hem definit segons la nostra configuració.
- ./condor-q: mostra la cua d'execucions, amb informació de les màquines que estan treballant, la màquina que ha enviat l'execució, el seu estat i el seu identificador.
- ./condor-status: mostra la relació de màquines treballadores i el seu estat en el moment de la consulta. Aquesta comanda ha sigut molt útil en la realització de l'algorisme mixt que proposo com a evolució.
- ./condor-rm: aquesta comanda, seguida d'un identificador de *job,* elimina el *job* de la cua.

2.1 Anàlisi Funcional del *GNU Chess*

2.2.1 Què es GNU Chess*?*

GNU Chess es un dels primers motors d'escacs desenvolupats pel projecte GNU per a

la plataforma *Linux*.

Una de les seves ultimes versions, la versió 5, va reescriure completament el programa utilitzant noves estructures de dades , i va incloure un nou motor de cerca anomenat *Principal Variation Search*.

La seva condició de software lliure[4], programat amb C, amb un algorisme de cerca conegut, i de arbre de profunditat a escollir pel jugador, el fa perfecte per a modelar-lo al nostre gust i fer un estudi sobre ell.

2.2.2 Interfície del GNU Chess

GNU Chess està dissenyat més com a eina d'estudi que com a element d'oci. Corre sobre consola i les opcions d'usuari són força limitades i poc intuïtives. Però dés d'un punt de vista algorítmic manté un motor de cerca força potent i molt interessant. Tanmateix se li calcula un Elo[5] de 2012, que el converteix en un rival força incomode per a algun jugador experimentat.

Des d'un punt de vista de jugador d'escacs, també contempla una opció força interessant, que permet carregar partides ja començades i guardades en un fitxer de text. D'aquesta opció ens servirem per fer les proves.

GNU Chess suporta el protocol UCI (*Universal Chess Interface*), que permet utilitzar una interfície gràfica per sobre del motor com *Xboard* (o *Winboard* si tenim un Sistema Operatiu *Windows*), per a poder fer les partides d'una forma més còmode.

4 *es pot descarregar a la pàgina : http://www.gnu.org/software/chess/chess.html*

5*'Elo' és un sistema de puntuació proposat per el fisic húngar Arpard Elo, que s'utilitza per a calcular la destresa de un jugador d'escacs (tant humà com 'virtual') . El rang de valors oscila entre 1000 (jugador aficionat) i 3001 (motor de búsqueda Rybka 2.2-x64) Un jugador amb un Elo de 2000 es considera un expert, i superar un Elo de 2400 et converteix en GM dels escacs. El primer jugador en superar un Elo de 2800 va ser Gary Kasparov el gener de l'any 2000. Més informació: http://en.wikipedia.org/wiki/Elo_rating_system*

Figura 3: *caputra de pantalla d'una partida en GNUChess*

Com es pot veure en la figura, encara que no es tracta d'un joc molt 'amigable', la quantitat de dades que rebem del còmput de la jugada per part del ordinador es considerable.

2.2.3 Motor de cerca del GNU Chess

Ja hem apuntat anteriorment que no es la nostra intenció fes un assaig complert sobre el programa, sinó fixar-nos en les característiques que ens poden servir per a poder contrastar bé el paradigma de la computació paral·lela i la computació 'tradicional' o seqüencial. Aquest apartat es basarà en l'estudi del algoritme del *Min-Max* i en com l'aplica el *GNU Chess* per treure'n el màxim rendiment. Aquest estudi és la base del projecte, ja que a partir d'ell entenem el com i el perquè s'han fet les modificacions posteriors. Començaré fent un recordatori del algorisme *Min-Max*, per a poder fer després l'explicació del PVS (*Principal Variation Search*), que es l'algorisme

que utilitza el programa.

Finalment explicaré com *GNU Chess* aplica el PVS, tant en l'apartat de cerca com en l'apartat de l'ordenació.

2.2.3.1 *Min-max* amb poda *alpha-beta*

Els jocs d'estratègia / lògica tenen en comú que segueixen unes normes molt definides, de manera que som capaços de conèixer tots els nostres pròxims moviments, així com els de l'adversari. Aquests moviments es poden representar en un arbre on cada jugada es representa en un nivell de l'arbre més profund que la jugada anterior. Lògicament, el nombre de nodes d'aquest arbre creix de manera exponencial i s'ha de definir una profunditat màxima.

L'algorisme del *Min-Max* es basa en donar una puntuació a cada estat del joc, aplicant, per exemple un valor positiu quan la partida està de part de l'ordinador i un valor negatiu quan està de part del jugador. Quan més allunyat és aquest valor de 0, la partida està més decantada per una banda o per l'altra.

Per aconseguir això, fa una cerca en profunditat de l'arbre i, quan arriba al nivell més profund, aplica una funció heurística definida prèviament a la 'fulla'. Al tornar al nivell superior, si suposem que correspon a una jugada de l'ordinador, òbviament es quedarà amb el màxim del valor de les fulles (ja hem dit que un valor positiu correspon a una partida decantada cap a l'ordinador) , i aplicarà al node aquest valor. Posteriorment, al tornar a un nivell més amunt, correspondrà a un torn del jugador, i en aquest cas es quedarà amb el mínim dels seus fills ja que *Min-Max* considera que el jugador optarà per la millor opció... i així successivament fins a arribar al node 'arrel'.

La figura 4 mostra el *pseudocodi* de l'algorisme del *Min-Max*, així com un exemple del

tractament sobre un arbre.

```
function minimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    else
        let α := -∞
        foreach child of node
            let α := max(α, -minimax(child, depth-1)
        return α
```

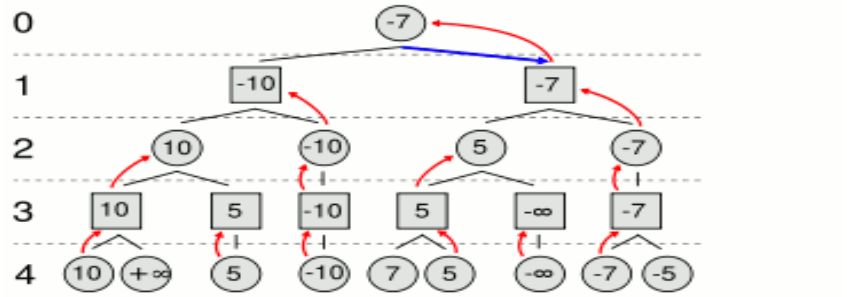


Figura 4: *Exemple de Min-Max*

La **poda alpha-beta** és una evolució del algorisme del Min-Max per a millorar la rapidesa del algorisme a costa de 'podar' branques de l'arbre. Aquesta optimització no afecta al resultat final del programa. Estableix una 'finestra' de valors que queden definits entre un valor mínim α i un valor màxim β que comencen com a -INFINIT i INFINIT respectivament i que es van acotant a mesura que avancem en l'algorisme. Quan tenim un valor α que sobrepassa β considerem que no fa falta avaluar més branques de l'arbre. És una bona idea definir una forma recursiva de codi que apliqui l'algorisme d'una manera simètrica com el següent exemple:

```
function alphabeta(node, depth, α, β)
    (* β represents previous player best choice - doesn't want it if α would worsen it *)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    foreach child of node
        α := max(α, -alphabeta(child, depth-1, -β, -α))
        (* use symmetry, -β becomes subsequently pruned α *)
        if β≤α
            break                               (* Beta cut-off *)
    return α
```

Figura 5: *Pseudocodi MinMax amb poda alpha-beta*

Aquest exemple va alternant els valors de α amb β per crear un algorisme amb un sol cas d'us alhora de podar: quan β es menor a α. És un bon exemple de pseudocodi perquè *GNU Chess* utilitza aquesta estratègia.

A continuació posarem un exemple per deixar clar perquè s'utilitza aquesta estratègia. Considerem el següent arbre a analitzar:

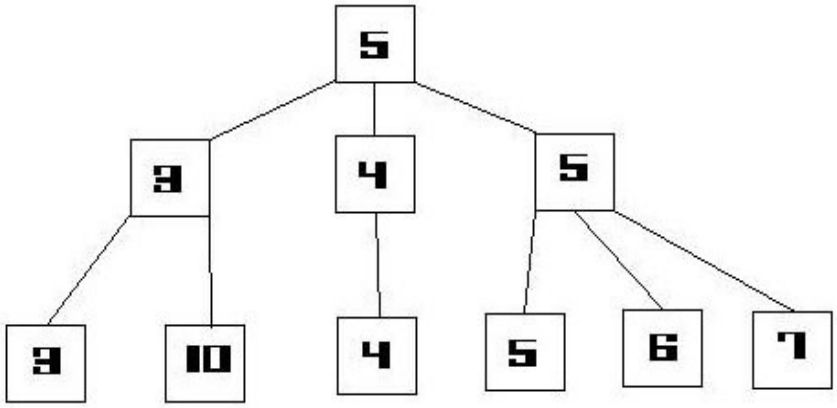


***Figura 6a**: Arbre exemple MinMax*

I que estem en aquest punt:

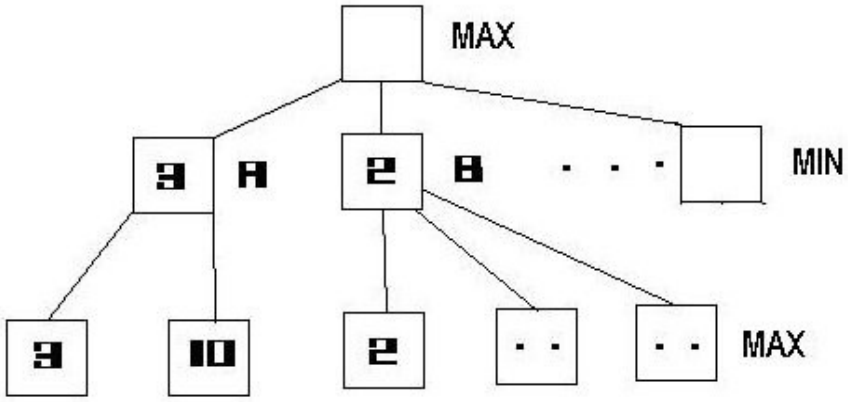


***Figura 6b:** Exemple MinMax*

Arribats al punt que acabem d'analitzar el primer fill del node B tenim un resultat de '-2' (rebem el resultat canviat de signe donada la simetria de l'algorisme que hem

definit abans) i que el valor β que tenim provinent del sub-arbre A és '-3'. Per tant, r*esultat* >= *β* i no farà falta seguir avaluant la resta de fills de B.

Això te una explicació molt lògica: Estem avaluant un node MIN i per tant, la resta de fills de B actualitzaran B de manera que el valor resultat serà mes petit o igual a 2. Serà inútil seguint-ho avaluant, ja que A té un valor més gran que B i el pare dels dos avalua un MAX. Això vol dir que sigui quin sigui el resultat de B, l'escollit sempre serà A.

2.2.3.2 Principal Variation Search (PVS)

La poda *alpha-bet*a mostra la peculiaritat següent: els millors moviments són els que acoten més la finestra entre α i β, i això significa que els nodes posteriors es podaran amb més facilitat. Si el millor moviment és el primer que s'avalua, la quantitat de branques podades serà màxima, i el temps d'avaluació mínim.

D'aquesta característica es nodreix el PVS (*Principal Variation Search*), que no és res més que un *Min-Max* amb poda *alpha-beta*, però que dona per suposat que el primer moviment avaluat és el millor i intenta avaluar els següents amb una finestra *buida* , de manera que intenta podar-los per sistema. Si la prova falla, llavors crida la resta de fills com un *Min-Max* normal.

```
function negascout(node, depth, α, β)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    b := β                                          (* initial window is (-β, -α) *)
    foreach child of node
        a := -negascout (child, depth-1, -b, -α)
        if a>α
            α := a
        if α≥β
            return α                                (* Beta cut-off *)
        if α≥b                                      (* check if null-window failed high*)
           α := -negascout(child, depth-1, -β, -α)  (* full re-search *)
           if α≥β
               return α                             (* Beta cut-off *)
        b := α+1                                    (* set new null window *)
    return α
```

Figura 7: *Pseudocodi del PVS (també anomenat Negascout)*

La utilitat depèn d'aquest algorisme depèn, doncs, de que l'arbre a analitzar estigui ordenat prèviament segons les expectatives de que els primers nodes a analitzar siguin els que generen els millors moviments. D'altra manera ens trobarem amb un algorisme que fa contínuament 'falles' (mirar el comentari *check if null-window failed high*) i ha de reprocesar dues vegades cada node, convertint-se en un algorisme menys òptim que el *Min-Max*.

GNU Chess utilitza una combinació de aquest algorisme amb un mètode de ordenació per a realitzar les cerques del millor moviment. La combinació l'ordenació amb el PVS es veu en el pròxim punt

2.2.3.3 Aplicació del *PVS* al *GNU Chess* (PVS + Iterative Deepening)

Per aconseguir un rendiment òptim, el *GNU Chess* combina un mètode de ordenació, un mètode de generació de nodes dinàmic i el PVS com a algorisme de cerca.

El bucle principal del programa fa tantes crides al algoritme de cerca com profunditat té l'arbre, cada vegada arribant a un node més profund. Aquesta tècnica s'anomena *Iterative Deepening*[6]

La idea es actualitzar a cada iteració els valors associats a cada node de l'arbre, de manera que agafem cada vegada un camí inicial (l'anomenat *Principal Variation*) més optim, per tal de millorar la rapidesa de la cerca i suavitzar l'increment de complexitat que comporta afegir un nivell de profunditat més gran al arbre.

La ordenació dels nodes es fa dinàmicament segons es van cridant els fills i sempre a la iteració següent a la creació i avaluació dels nodes.

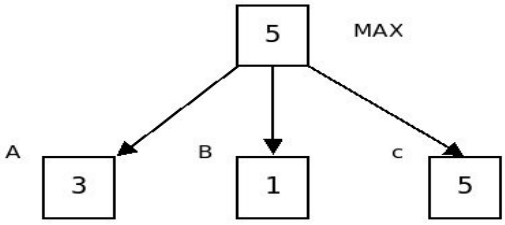
---

6 *Podem trobar informació a la wikipedia: http://en.wikipedia.org/wiki/Iterative_deepening i en el següent PDF, molt recomanable com a repàs i estudi de algorismes de cerca: http://homepages.cwi.nl/~paulk/theses/Carolus.pdf , encara que no deixa aquesta tècnica massa ben parada.*

Tot això es combina amb el mètode de cerca en profunditat de l'arbre PVS que hem vist en l'anterior punt.

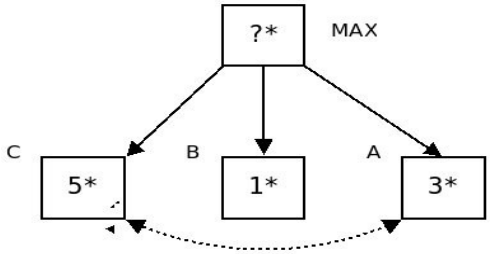
Per entendre la ordenació combinada amb l'*Iterative Deepening*, serà més fàcil de veure-ho en un exemple:

Considerem els nodes A, B i C, que s'han creat a la primera iteració i han agafat els valors 3, 1 i 5 respectivament. Com que es una sola iteració i en són les fulles, s'han avaluat amb una funció heurística.



***PVS pas 1**: execució de la primera iteració.*

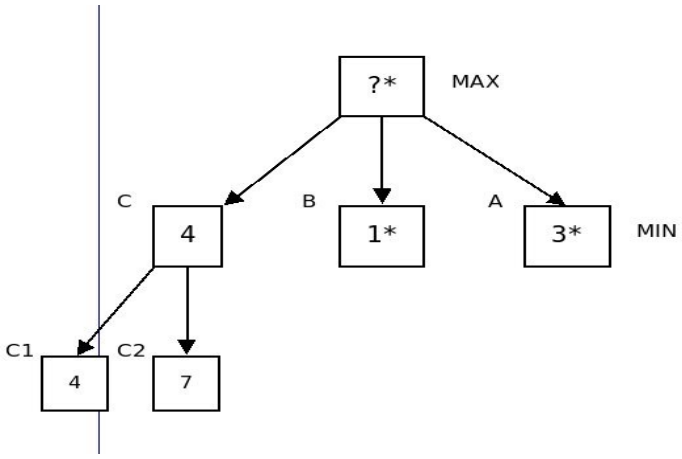
A la següent iteració crearem un arbre de profunditat dos i començarem escollint el primer node a través de la funció de ordenació. Aquesta funció es basarà en els valors de l'anterior iteració. Adjuntaré un '*' per mostrar que no es tracta de valors de la actual iteració.



***PVS pas 2**: swap de nodes no-fulles segons el seu valor anterior*

Veiem, dons que escull el node C en primer lloc. El següent pas que fa és el de crear els nodes fill, avaluar-los mitjançant la funció heurística i actualitzar el valor de C (ara

ja no tenim un valor amb * sinó l'avaluació del MIN dels fills C1 i C2) :

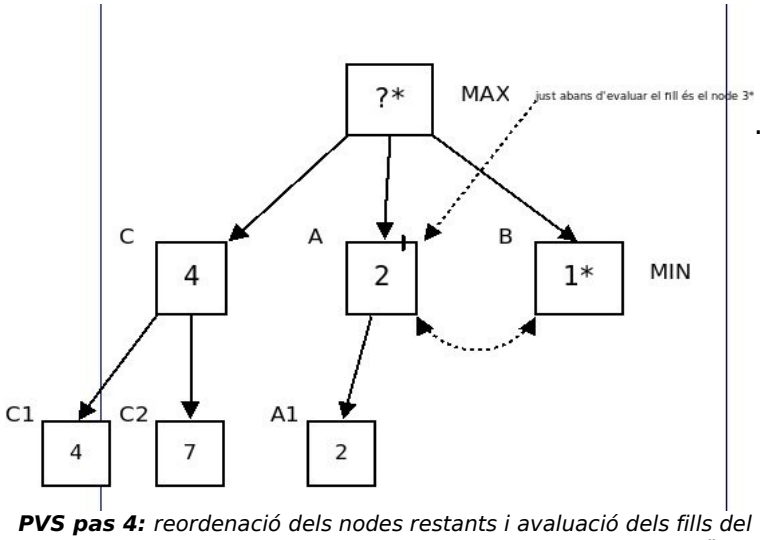


***PVS pas 3**: avaluació del MIN del primer node*

Llavors escull el següent node a avaluar. De nou l'algorisme d'ordenació ha d'escollir quin és el major dels nodes avaluats en la anterior iteració que té un valor més alt. Ens trobem que de nou fa *swap* entre els nodes A i B, ja que el node A té un valor més gran que el B.

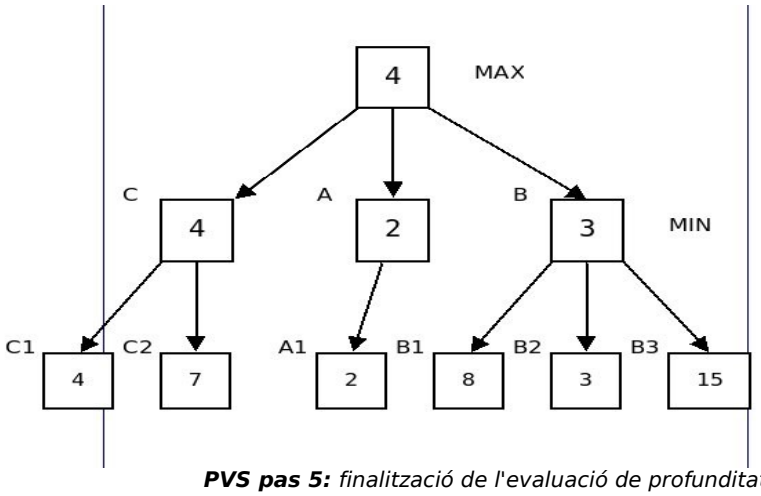
El que farà després és el mateix que en el node C: genera els fills del node A. En aquest cas, per economitzar espai en el gràfic, hem posat com a exemple que només genera un fill. De totes maneres, ens imaginem que avalua el Min i actualitza el valor de A.

Podem veure aquests passos en la figura *PVS pas 4* que mostrem a continuació en la següent pagina:



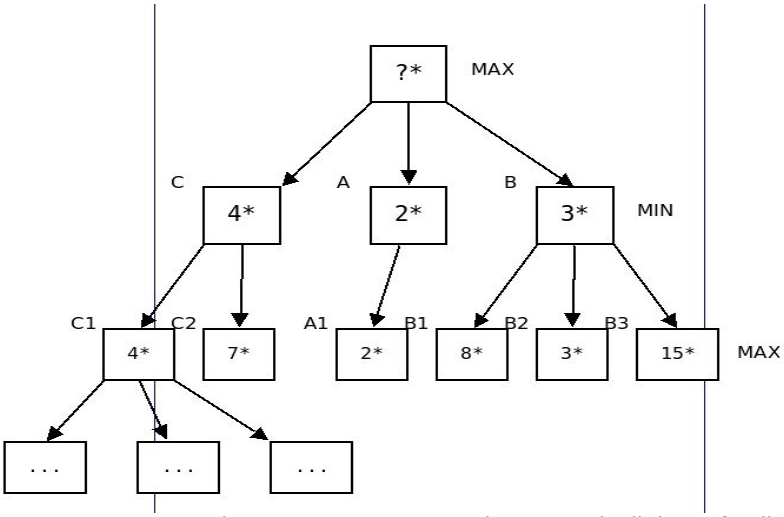
PVS pas 4: *reordenació dels nodes restants i avaluació dels fills del següent*

Després fa lo propi amb el node B, estenent els seus fills i actualitzant el valor associat al seu node. En aquest cas, per avançar feina, ja hem inclòs l'avaluació del MAX del node arrel en el gràfic:



PVS pas 5: *finalització de l'evaluació de profunditat 2*

Ara tocaria tornar a fer tot el recorregut per la tercera iteració.. i així fins a la profunditat màxima establerta per el jugador...



PVS pas 6: *i tornem a començar amb un nou nivell de profunditat...*

En el cas anterior cal fixar-se que no reordenem els nodes C1 i C2 encara que C2 sigui més gran. Això ho fem perquè en aquest cas la ordenació és inversa, ja que el seu pare (C) vol avaluar el millor node primer, i aquest millor node és el que té un valor més petit. Més endavant, però caldrà reordenar de nou els sub-arbres A i B, ja que B presenta un valor més alt que A en la iteració anterior.

Creiem que d'un mode gràfic ja hem donat a entendre el qu fa l'algorisme d'ordenació. No ens hem d'oblidar que apart d'això, utilitza el PVS per recórrer l'arbre i que algunes d'aquestes branques es podaria abans de avaluar-se. Crec que es el moment de plasmar la mateixa idea, però a través d'un pseudocodi.

Un pseudocodi de l'algorisme iteratiu que utilitza el *GNU Chess* podria ser aquest:

```
function iterate(prof_max) is

      alpha = -INF;
      beta = INF;
```

```
    int prof_act = 1;
    mentre prof_act <= prof_max
        pvsRoot(node_root, prof_act, alpha, beta);
        actualitza_alpha_beta();
        prof_act++;
    fi_mentre;
    return node_root->moviment;
fi_function
```

```
function pvsRoot(node, profunditat, alpha, beta) is
    mentre teMesFills(node)
        fill = seguentFill(node);
        valor = pvs(fill, profunditat, alpha, beta);
        si (valor > millor) llavors
            millor = valor;
            node_root->moviment = fill->moviment;
        fi_si
    fi_mentre
fi_function
```

```
function pvs(node, profunditat, alpha, beta)
    si profunditat==1 llavors
        return heuristica(node)
    fi_si
    fills = obté_fills(profunditat, node)
    b = beta
    per n=1 a n=num(fills) fer
        fill_actual = agafaSeguent(n,fills);
        a = pvs(fill_actual, profunditat-1, -b, -alpha)
        si a > alpha
            alpha = a
```

```
            fi_si
            si alpha > beta llavors
                  return alpha
            fi_si
            si alpha > b llavors
                  alpha = pvs(fill_actual, profunditat-1, -beta, -alpha)
                  si alpha > beta llavors
                        return alpha
                  fi_si
            fi_si
            b = alpha + 1
      fi_per
      return alpha
fi_function
```

```
function obte_fills(profunditat, node)
      si profunditat == 2 llavors
            fills = generaFills(node)
            per n=1 fins n=num(fills)
                  fills[n]->valor = assignaValorArbitrari()
            fi_per
            return fills
      fi_si

      sino return recuperaPunterAmbFills(node)

fi_function
```

```
function agafaSeguent(n,fills)
      long = num(fills)
      millor = fills[n]
      per i = n+1 fins i =fills[long] fer
```

```
            si fills[i] > millor llavors
                  millor = fills[i]
            fi_si
      fi_per
      intercanviaValors(fills[n], millor);
fi_function
```

Un dels temes que queda pendents d'estudi, i bàsicament per falta de temps, seria el que correspon a la funció `recuperaPunterAmbFills(node)` que s'encarregaria de gestionar la memòria alhora de recórrer l'arbre i jugar amb els punters de l'aplicació per obtenir els fills de un determinat node que ja ha estat avaluat en una iteració anterior. Després de fer un intent d'entendre la gestió durant forces moments 'morts' de totes les hores que he dedicat al projecte, tinc motius per pensar que aquest seria un tema que ens donaria per un altre projecte de recerca sencer.

Al final vaig decidir desistir ja que considero que tot i ser un tema molt interessant per entendre la potencia del algorisme, no crec que guardi massa relació amb el tema que ens ocupa: l'adaptació a un sistema d'execució paral·lela.

2.2.3.4 Estudi del rendiment del motor de búsqueda

Un cop analitzat el mètode de cerca del *GNU Chess*, és molt fàcil que un es qüestioni l'estratègia escollida alhora de recórrer l'arbre: És realment un algorisme eficient? Si escollim una profunditat 10, és necessari reconstruir l'arbre 10 vegades encara que el tinguem cada cop més ordenat i el PVS sigui molt eficient en aquests casos? No seria més senzill crear-lo un sol cop i aplicar un simple *Min-Max* amb *alpha-beta* i així inclús simplifiquem funcionalment el programa?

Aquests dubtes van sortir durant el desenvolupament del projecte, així com crec que

poden haver sortit en qualsevol que hagi llegit l'apartat anterior. Per això he cregut necessari incorporar un apartat de estudi del algorisme per sortir de dubtes.

Una estratègia *Iterative Deepening* genera una vegada els nodes del nivell més baix de profunditat, dues vegades els nodes de profunditat-1, tres vegades els nodes de profunditat-2.. etc. .. fins a l'arrel del sistema, que s'expandeix (profunditat+1) vegades. Això es defineix per la formula:

$(d+1)1 + d*b + (d-1)b^2+(d-2)b^3+ ... + 2bExp(d-1) + bExp(d)$

on d = profunditat, i b= numero de fills de cada node.

Si tenim un arbre de profunditat 4 i el *Branching Factor* (nombre de fills de cada node, de mitjana) dels jocs d'escacs es mou entre el 30 i el 35 segons on es consulti (Agafarem 35), llavors es generarien:

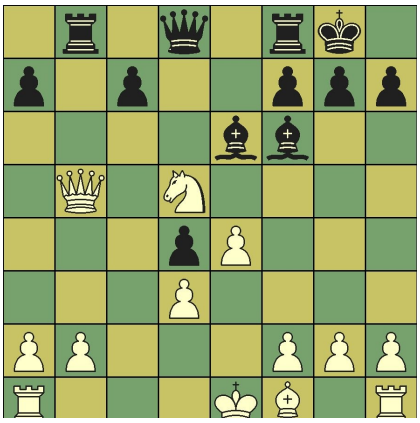
$5 + 4*35 + 3*(35*35) + 2*(35*35*35) + (35*35*35*35) = 5 + 4*35 + 3*1225 + 2*42875 + 1500625 = 5 + 140 + 3675 + 85750 + 1500625 =$ **1590195 nodes**

Aquesta fórmula suposa que *Iterative Deepening* genera un 9.4% més de fills que estendre l'arbre fins a la profunditat màxima una sola vegada (**1500625 nodes**). Si la profunditat tendeix al infinit, el percentatge de fills que genera aquesta tècnica en respecte a la 'normal' tendeix al 11%[7]. Queda veure doncs, si en un arbre degudament ordenat, el PVS aconsegueix rectificar aquesta desavantatge amb podes massives de branques i en conseqüència, en la no generació de sub-arbres que no ens porten enlloc.

Més endavant veurem el joc de proves que s'ha utilitzat en aquest projecte. Es tracta de 9 moments en una partida d'escacs i dels quals s'avaluarà la resposta en les solucions proposades. Hem escollit un d'aquests moments de la partida,

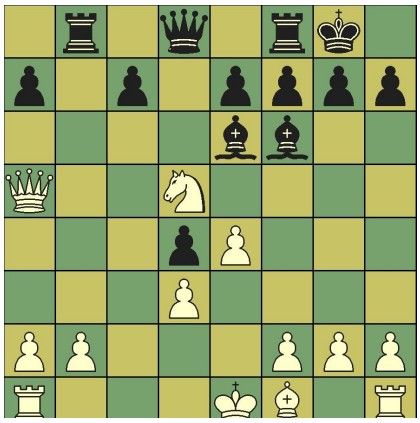
7 *No ho dic jo, ho diu la wikipedia!* http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search.

possiblement el més determinant i el que necessitava de més temps per ésser avaluat per a fer la prova de rendiment que ens ocupa. Agafarem aquest moviment per comparar el PVS+*Iterative Deepening* vs el *Min-Max*. Ens trobem en aquest moment de la partida:



I nosaltres, intentant fer servir la nostra lògica alhora de jugar a escacs, encara que es podria haver escollit qualsevol altre cosa, fem el següent moviment:

Qa5



La idea per provar el rendiment de l'algorisme és la següent: Nosaltres tenim que el pseudocodi per al PVS que fa una crida `a = pvs(fill_actual, profunditat-1,`

`-alpha-1, -alpha)` de manera accedeix recursivament a una finestra nul·la. La condició per comprovar si la prova de la finestra nul·la ha fallat es redueix a '*beta>a>alpha*' . Si aquesta condició es dóna, farem la crida 'normal' `a = pvs(fill_actual, profunditat-1, -beta, -alpha).` No serà difícil doncs, convertir l'algorisme PVS al *Min-Max* normal: La primera crida es convertirà a `a = pvs(fill_actual, profunditat-1, -beta, -alpha)` i comentarem la segona, per no cridar dos cops a la mateixa funció de la mateixa manera.

Per altra banda, assignarem directament la profunditat màxima a la primera iteració perquè només generi l'arbre una vegada, eliminant l'*Iterative Deepening*. Utilitzarem un arbre de profunditat 7. El resultat es pot veure en aquesta captura de pantalla:

***Figura 8**: Avaluació del Min-Max*

La taula del principi mostra la profunditat , el primer node avaluat i que l'estem expandint (els valors 7& -> *Rxb2*). Després veiem que és un bon candidat el moviment *Bxd5* (*7+*), l'expandim (7& -> *Bxd5 Qxd5*) i el confirmem (7.). Més o menys o podem llegir així.

Les dades que surten a continuació corresponen a la quantitat de nodes que es generen (*GenCnt*) , que s'avaluen (Nodes), que es poden (*NullCut*)... etc. I finalment tenim l'estat de la partida i el node escollit.

Figura 9: *Exemple de PVS amb Iterative Deepening*

Per fer més evident l'avaluació també s'ha modificat la sortida incorporant una línia explicativa dels nodes generats i avaluats. **S'han calculat 9.888.330 nodes dels 17.034.092 generats**. També és interessant observar la línia "**My elapsedTime**

is”, que correspon a **21.21** segons per avaluar el moviment.

Ara és torn de tornar al PVS original i avaluar la seva resposta. Utilitzarem la mateixa profunditat i el mateix moment de la partida que l'exemple anterior i aquest és el resultat:En aquest cas s'ha generat una línia de nodes avaluats/generats per a cada iteració de profunditat de l'arbre.

El primer que ens hem de fixar és que el moviment de resposta i l'avaluació que li dóna al moviment són la mateixa en els dos algorismes: Bxd5 (amb el seu alfil em mata el meu peó de la casella d5) i li assigna un valor de 220. Per tant, en cap dels dos algorismes perdem eficiència de càlcul, els dos fan bé la seva feina. La diferència arriba en la rapidesa i nombre d'operacions que necessitem per arribar a aquest resultat. Per fer més entenedora la comparativa, s'ha creat el següent quadre basant-nos en les captures de pantalla:

	Nodes Calculats		**Nodes Generats**		**Temps de Càlcul**		**Resultat**	
	MinMax	PVS	MinMax	PVS	MinMax	PVS	MinMax	PVS
Profunditat 1	0	95	0	410				
Profunditat 2	0	648	0	1.618				
Profunditat 3	0	2.239	0	4.915				
Profunditat 4	0	27.235	0	46.704				
Profunditat 5	0	147.918	0	22.785				
Profunditat 6	0	227.575	0	319.080				
Profunditat 7	9.888.330	458.476	17.034.092	662.861				
Total	**9.888.330**	**864.186**	**17.034.092**	**1.058.373**	21,21	0,83	Bxd5->220	Bxd5->220

Esquema 1: *PVS + Iterative Deepening vs MinMax*

Podem veure doncs, que l'estratègia utilitzada pel *GNU Chess* es totalment encertada i els dubtes sobre l'eficiència del algorisme s'esvaeixen en un moment: primer de tot fixant-nos en el temps de càlcul, 21.21 per el *Min-Max*, contra 0,85 del *PVS+Iterative Deepening*. Aquesta diferència també es fa patent en el nombre de nodes que necessita: en aquest moviment en concret el PVS genera un 6,2 % dels nodes que genera el *Min-Max* un i calcula 8.7% dels nodes que calcula el *Min-Max*. La computació, per tant, és molt més òptima.

Això també ens porta a una altra conclusió: tenir un arbre ordenat implica que els

moviments que generem són més susceptibles de ésser calculats. Mentre en el PVS només es descarta un 19.4% dels moviments que hem generat i es calculen tots els altres, en el cas del *Min-Max* aquest valor arriba fins al 42%.

Aquesta conclusió ens agrada, però també ens fa respecte: a partir d'ara el repte serà millorar de forma paral·lela un algorisme que està pensat per treballar de forma seqüencial i de manera molt òptima en una sola màquina.

3. GNU Chess Paral·lel. Canvis realitzats.

Un cop arribats a aquest punt toca explicar el desenvolupament que s'ha portat a terme en el projecte. Aquest es un punt on es fusionen els coneixements desglossats en l'apartat 2 de la memòria i s'apliquen per tal d'arribar a un dels objectius marcats: aconseguir una aplicació que s'executi paral·lelament sense perdre informació i arribar a conclusions contrastant l'anterior apartat amb el desenvolupament d'aquest.

Aquest apartat es divideix en tres blocs: en el primer, farem una reflexió del que tenim i el que volem aconseguir, en el segon explicarem en mode esquemàtic quina es la nostra idea alhora de dividir la feina per enviar-la a *Condor*. I en el tercer repassarem punt per punt els canvis realitzats, entrant en estructures de dades que val la pena tenir en compte i els canvis més significatius en les línies de codi, intentant explicar en tot moment el perquè d'aquests canvis.

3.1 Reflexió sobre la computació en paral·lel.

Una de les primeres coses que hem de tenir en compte és que l'algorisme PVS + *Iterative Deepening* està plenament optimitzat per a l'execució seqüencial en una cerca en profunditat. Un dels punts forts d'aquest algorisme és que, si tenim l'arbre ordenat i calculem el primer fill del node arrel, obtenim uns valors *alpha-beta* que podran podar gran quantitat de nodes en la resta de fills.

Si creem una aplicació paral·lela que envia cada un dels fills a executar-se en paral·lel a través de *Condor*, la característica anterior no es pot donar, perquè el segon fill desconeix els valors *alpha-beta* obtinguts en el primer. Per tant, guanyem potencia de càlcul, però perdem 'intel·ligència' d'algorisme i calculem molts més nodes.

Per altra banda, hem de tenir en compte que la configuració de *Condor* en el Gestor Central retarda l'enviament de treballs en un ordre de 1 o 2 minuts. Tot i que nosaltres en els resultats no tinguem en compte aquesta característica i avaluem el resultat únicament per el temps de la seva execució ja que no depèn de nosaltres la configuració de *Condor*, hem de ser conscients que fer un *Iterative Deepening* i enviar una cua d'execucions per a cada iteració és senzillament inviable per a qualsevol persona, sigui quina sigui la seva paciència.

Si hem d'afegir més problemes a l'execució paral·lela amb aquest algorisme, també hem de considerar que tot i que no acotem els valors *alpha-beta* en cada un dels fills d'una iteració, si que ho fem entre iteració i iteració. A la enèsima iteració tindrem un arbre força ordenat amb uns *alpha-beta* força acotats. Això significa que enviarem molts treballs a *Condor* que simplement es podaran a la primera, es a dir, una càrrega a la xarxa i a la cua de *Condor* totalment innecessària, sense tenir en compte el retrasament de l'execució.

Conclusió: ja que tirem de força bruta, fem-ho bé. L'algorisme que proposem a continuació és simplement un arbre Min-Max que s'analitza de forma exhaustiva i es desplega en la seva màxima profunditat una sola vegada, a diferència de l'I*terative Deepening*, que es desplegava N vegades, on N es la profunditat definida. Al desplegar-ho un sol cop, tindrem un arbre desordenat que només s'enviarem una cua de treballs a *Condor.* D'aquesta manera aconseguirem que la càrrega de la xarxa i la cua de *Condor* s'aprofitin calculant un gran nombre de nodes a cada màquina remota.

3.2 Esquema gràfic i algorítmic de la nova aplicació.

L'estructura de la nova aplicació es pot explicar recordant el pseudocodi de l'anterior apartat. En concret aquesta funció:

```
function  pvsRoot(node, profunditat, alpha, beta) is
     ment re  teMesFills(node)
          fill = seguentFill(node);
          valor = pvs(fill, profunditat, alpha, beta);
          si  (valor > millor) llavors
               millor = valor;
               node_root->moviment = fill->moviment;
          fi_si
     fi_ment re
fi_function
```

Si imaginéssim que '*pvs*' és una aplicació externa al programa que executa *pvsRoot*, i es crida a través de *Condor*, ja tindríem la idea de com funciona.

Però intentarem ser una mica més fidels a l'aplicació creada, primer creant un gràfic amb la idea del treball de *Condor* aplicat al *GNU Chess* i després modificant el pseudocodi anterior i adaptant-lo al nou desenvolupament. Per últim farem una explicació narrativa del procés.

Les màquines i fitxers que intervenen en el procés es poden veure en el següent esquema.

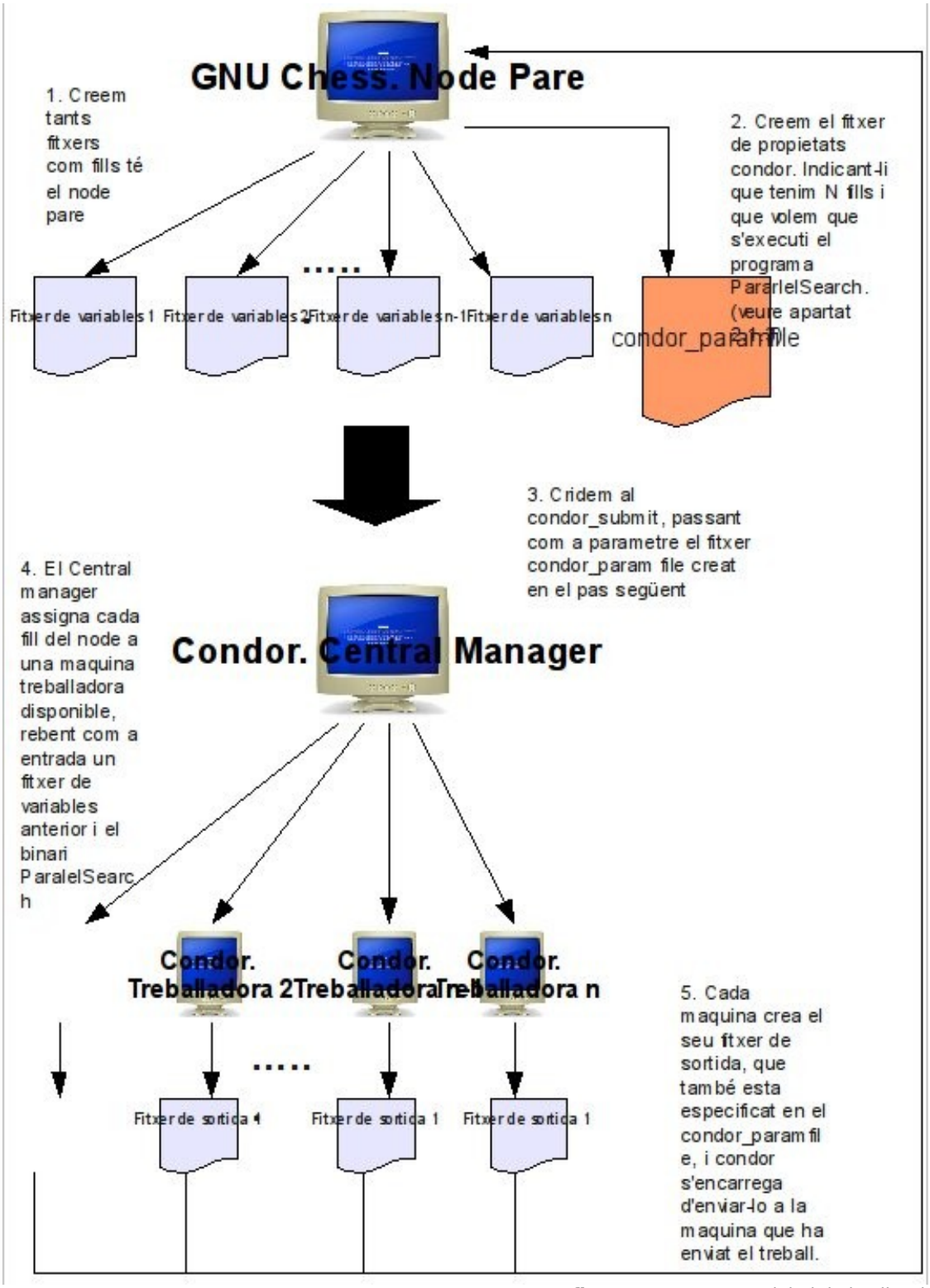


figura 10: *esquema global de l'aplicació*

Ara que tenim una idea de el que intentem fer, podem entrar en el pseudocodi per

mostrar les diferents parts d'una manera algorísmica i poder-lo comparar amb el *pvsRoot* anterior.

```
function  pvsRootParalela(node, profunditat, alpha, beta) is

     ment re teMesFills(node)
          fill = seguentFill(node);
          preparaFitxerCondor(fill, profunditat, alpha, beta);
     fi_ment re
     cridaCondor();
     tornaIniciNode(node);
     ment re teMesFills(node)
          si creem_Thread = fill llavors
               node->valor = esperaExecucióCondor()
          fi_si
          sinó
               continua();
          fi_sinó
     fi_ment re
     tornaIniciNode(node)
     ment re teMesFills(node)
          si (node->valor > millor) llavors
               millor = node->valor;
               node_root->moviment = node->moviment;
     fi_ment re
fi_function
```

Per començar tenim una aplicació que la seva funció principal es el PVS (l'anomeno *ParalelSearch* com en el codi 'real'). La idea es cridar-la amb totes les variables que necessita per a l'execució. La quantitat de variables que es necessitaven va resultar ser massa gran per passar-ho per paràmetres (més de cent), i es va decidir fer-ho a través d'un fitxer.

El primer bucle '**mentre**' es dedica a crear un fitxer amb cada node 'fill' que necessitem, de manera que tindrem preparat un fitxer de variables per a cada execució.

Llavors, la funció *cridaCondor()* prepara una cua d'execucions, tantes com fills té el node arrel, i crida a la comanda *condor_submit* per començar l'aplicació.

Després, creem un thread per a cada execució que estem esperant, extraiem la variable que necessitem del fitxer de sortida corresponent a cada maquina treballadora, i quan els tenim tots, avaluem quin es el millor moviment dels que ens han arribat. La idea és que tota aquesta execució sigui transparent al jugador que executa el joc.

Fins ara, en aquest pseudocodi hem vist la part que s'executa a la nostra màquina. En cada una de les màquines treballadores, s'executarà aquest:

```
function  ParalelSearch(File entrada)

     File sortida;

     llegirFitxer(entrada) //aqui ja actualitza les dades de
                              node, profunditat, alpha i beta
     valor = pvs(node, profunditat, alpha, beta)
     escriuFitxer(sortida, valor);

fi
```

On *pvs* s'executa exactament igual que el definit en l'apartat 2.2.3.3

3.3 Detall dels canvis realitzats.

Cada apartat a partir d'ara suposarà un anàlisi més concret de cada mòdul o conjunt funcional del desenvolupament.

3.3.1 Funcions de lectura i escriptura de fitxers

S'acaba de dir que tenim dos executables per el *GNU Chess* paral·lel: el *GNU Chess* anterior amb algunes modificacions i un *ParalelSearch* que bàsicament executa el PVS en una màquina remota i retorna uns valors amb l'avaluació del node. El pas de dades entre els dos executables es fa mitjançant fitxers que són gestionats com a entrada/sortida dels programes gràcies a *Condor.*

El fitxer de dades que escriu *GNU Chess*, anomenat *chess.param_N* (on N es l'enèsim fill del node arrel), inclou 110 variables, necessàries perquè quan *ParalelSearch* comenci a executar-se ens trobem exactament en el punt on hem tallat l'execució del *GNU Chess*. Creiem necessari destacar les següents variables/estructures:

- int score: enter que guarda el valor del millor moviment fins al moment.
- int RootAlpha/int RootBeta: enters que guarden l'*alpha-beta* inicial de la execució
- int Idepth: profunditat màxima del *Min-Max*
- double et: són les sigles de *Elapsed Time*. Ens interessa per guardar una relació del temps que tarda el programa a executar-se, alhora de treure conclusions i guardar als *logs*
- leaf *TreePtr[MAXPLYDEPTH]: aquesta es, possiblement la variable més important alhora de recórrer l'arbre per el *Min-Max* i la que ens va donar més problemes, ja que guarda valors dels nodes entre els índexs de l' *array*. '*Leaf*' és una estructura de dades que manté informació de un node i del seu

valor associat:

```
typedef struct
      {
        int move;
        int score;
} leaf;
```

La variable *TreePtr* és un *array* d'apuntadors a aquestes estructures, programat de manera que *TreePtr*[1] apuntarà al primer fill del primer nivell de nodes, *TreePtr*[2] apuntarà al primer fill del segon nivell de nodes, *TreePtr*[3] apuntarà al primer fill del tercer nivell de nodes.. etc...

Per tal de accedir a tots els fills d'un nivell, necessitarem un altre *leaf* *p, que recorri totes les direccions de memòria existents entre *TreePtr*[n] i *TreePtr*[n+1].

- Board board: La variable de *board* de tipus *Board* mostra un 'resum' de l'estat de la partida en cada moment i segueix la següent estructura:

```
typedef struct
{
      BitBoard b[2][7];
      BitBoard friends[2];
      BitBoard blocker;
      BitBoard blockerr90;
      BitBoard blockerr45;
      BitBoard blockerr315;
      int ep;
      int flag;
      int side;
      int material[2];
      int pmaterial[2];
      int castled[2];
      int king[2];
} Board;
```

on *Bitboard* equival a un enter de 64 bits que representa un tauler (8x8). Hi podem veure la informació de la posició de cada peça, la pròpia taula , amb les seves transposades, el temps que es porta de partida, el costat del que juga, si ja s'ha enrocat...etc.

Un cop escrit el fitxer de variables *chess.param_N*, arriba al *ParalelSearch*, que actualitza tota la informació de les variables, executa el seu codi, i escriu en un fitxer de sortida *search.param_N* (on 'N' ha de coincidir) el resultat de l'execució.
Aquest fitxer és molt més senzill que l'anterior i només hi escriurem els valors següents:

- moviment
- valor
- temps inicial i final
- *elapsed time* de l'execució
- nombre de nodes calculats

Com es pot veure, donat que el moviment ja el coneix l'aplicació *GNU Chess*, l'unic valor destacable que li enviem es el valor de l'avaluació del moviment. La resta també serviran per *logs* i conclusions.

Llavors, el *Condor* li tornarà al fitxer al *GNU Chess* i aquest tindrà N *threads* esperant l'arribada d'aquests fitxers, que els llegiran i continuaran amb l'execució.

3.3.2 Cridant a Condor

Un cop s'han preparat tots els fitxers per a l'execució remota, s'haurà de preparar un fitxer, que anomenarem *condor_submit* de la mateixa manera que s'ha explicat anteriorment en l'apartat 2.1.3. La funció que genera aquest fitxer, únicament li cal el nombre de fills que té el node arrel. Es pot tornar a posar com a exemple de fitxer generat el següent.

```
# Inici del fitxer /home/jdevesa/files/file_example
Universe = vanilla
Executable = /home/jdevesa/bin/ParalelSearch
```

```
Output = condor_submit_out_$(Process)

Error = condor_submit_err

Log = condor_submit_log

Arguments =  3 72 37 96 37 2 $(Process)

Transfer_input_files = chess.param_$(Process)

Transfer_output_files = search.param_$(Process)

InitialDir = /home/jdevesa/files/

Should_transfer_files = YES

When_to_transfer_output = ON_EXIT

Queue  30

# Final del fitxer /home/jdevesa/files/file_example
```

on es crida al executable *ParalelSearch*, se li indica quins son els fitxers d'entrada (que ja hem preparat) i els de sortida.

Després de crear aquest fitxer s'executa el codi corresponent a enviar peticions d'execució a *Condor*:

```
if(fork() == 0) {
      printf("PFC: Llamando a la cola de ejecucion!\n");
      exec_ok = execv("condor_submit",cmd);

}
else waitpid(-1,&status,0);
```

on *cmd* és el nom del fitxer *Condor* que acabem de crear. Això ja crearà els N treballs a la cua d'execució.

La crida **execv** acaba l'execució de l'actual aplicació i retorna 0. Per això, ha sigut necessari fer un *thread* pare i un *thread* fill, perquè un acabi i l'altre continuí amb la execució del *GNU Chess*. La comanda **waitpid** espera el final de l'execució del fill per continuar. (Espera que acabi de cridar el *condor_submit*, no que acabi l'execució del

Condor! Això s'ha de fer manualment!).

3.3.3 Sincronització de threads*.:*
Esperant l'execució de Condor*.*

Ja hem vist com es preparaven i quines variables tenien els fitxers d'entrada i sortida a l'aplicació que s'havia d'executar remotament, i com amb la comanda *condor_submit* i un fitxer de propietats enviàvem una cua d'execucions amb els fitxers d'entrada com a propietats. Ara ens queda per veure com s'ha fet perquè el *GNU Chess* rebi el resultat de les execucions a través dels fitxers de sortida.

Uns dels requisits que ens vam auto imposar per a què la aplicació tingués més bon rendiment, i que fos realment paral·lela, era que el programa ha d'estar preparat perquè les execucions remotes acabin en qualsevol ordre. Això vol dir que hem de tenir una forma *asíncrona* de esperar els fitxers, tenir certa informació dels que tenim i dels que ens falten per a seguir esperant o no, i amb això hem d'aconseguir que cada resultat s'apliqui al node que li correspon, independentment de l'ordre en que arribin.

Aquesta sincronització i espera dels fitxers es fa mitjançant *threads* i *pipes.*

S'ha creat una jerarquia de pares i fills de manera que el pare no es mor fins que llegeix el fitxer que se li ha assignat i posteriorment espera que el fill li passi el resultat de la seva execució. Quan el fill ha llegit el fitxer, espera si el seu corresponent fill corresponent ha acabat, llegeix les seves variables i, si es així, mor. L'únic que acaba amb vida és el primer dels *threads* que s'ha creat (el de més amunt en la jerarquia).

Encara que fet d'aquesta manera establim un ordre de morts, tots estan escoltant al mateix temps i el fet de passar informació al pare i morir a continuació és pràcticament immediat. Hem creat un esquema per a una execució paral·lela de 4

nodes.

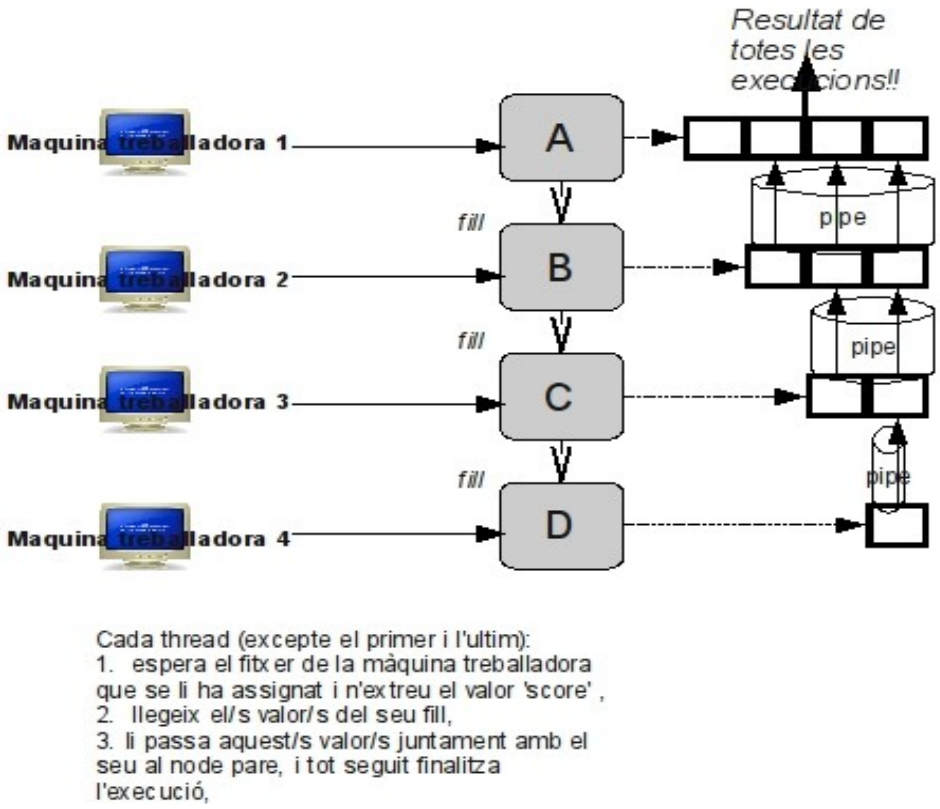


figura 11: *esquema de la recuperació de les dades dels fitxers de forma paral·lela*

La funció que s'encarrega d'aquesta lectura en paral·lel retorna un apuntador a enters que fàcilment es pot convertir en un *array* d'enters o millor dit al valor '*score*' un d'*array* de estructures de tipus *leaf* que ja hem explicat anteriorment i sincronitzat amb els valors de moviment associats.

M'explico millor, i per fer-ho recuperem el primer tros del pseudocodi modificat i també l'estructura *leaf:*

```
ment re  teMesFills(node)
           fill = seguentFill(node);
```

```
            preparaFitxerCondor(fill, profunditat, alpha, beta);
fi_ment re
```

```
typedef struct

{
        int move;
        int score;
} leaf;
```

el que en realitat tenim es un *array* anomenat `leaf *movim[MAXPLYDEPTH]`. Per a cada crida a la funció `fill = seguentFill(node);` el que anomenem 'fill' és en el programa una referència a un moviment que s'actualitza a *movim[i]->move*.

Hi ha també una variable global inicialitzada a 0 i incremental, que es concatena al final del fitxer d'entrada a l'aplicació remota i que ja hem dit que s'anomena *chess.param* i que augmenta cada vegada que creem un fitxer. Per tant, l'estat del joc del fitxer *chess.param_0* correspon al moviment *movim[0]->move*, l'estat del joc en el fixer *chess.param_1* correspon al moviment *movim[1]->move* i el *chess.param_N* correspon al moviment *movim[N]->move.*

Quan el fitxer de propietats del *Condor condor_submit* estableix aquesta regla:

```
Transfer_input_files = chess.param_$(Process)
Transfer_output_files = search.param_$(Process)
```

associa per a cada fitxer d'entrada d'una execució el seu fitxer de sortida amb la mateixa cardinalitat. Així, l'execució que té d'entrada el fitxer *chess.param_N* tindrà com a fitxer de sortida el fitxer *search.param_N.*

La funció que acabem d'explicar , llegeix paral·lelament els fitxers *search.param_**, de manera que aconseguim ficar-ho en un *array* d'enters anomenat `int`

`array_score[MAXPLYDEPTH]`. Aquest *array* conté per a cada un dels seus index **i** el valor resultat de l'execució emmagatzemat al fitxer *search.param_***i** .

El següent pas és obvi. Fem una assignació **movim[i]->score = array_score[i]** de manera que cada moviment té el seu valor associat i preparats per a ésser avaluats.

En aquest cas, donat que tenim un codi totalment nou i no es tracta d'una funció amb masses línies de codi i força important per la correcta execució paral·lela del programa, la inclourem en la memòria.

```
/* PFC GNU Chess 5.1 Beta: Carreguem les variables del fitxer     */
/* GNU Chess 5.0 - iterate.c - iterative deepening code
   Copyright (c) 1999-2002 Free Software Foundation, Inc.

   GNU Chess is based on the two research programs
   Cobalt by Chua Kong-Sian and Gazebo by Stuart Cracraft.

   GNU Chess is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2, or (at your option)
   any later version.

   GNU Chess is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with GNU Chess; see the file COPYING.  If not, write to
   the Free Software Foundation, Inc., 59 Temple Place - Suite 330,
   Boston, MA 02111-1307, USA.

   Contact Info:
     bug-gnu-chess@gnu.org

   Canvis Projecte Final de carrera (PFC) per Jaume Devesa.
   comentats com a 'PFC GNU Chess 5.1 Beta'
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <unistd.h>                     /* PFC -- necesaria la funcion execp
                                */
#include "common.h"
```

```
FILE *prm_f;                                                    // fitxers
de parametros
char buffer[200];                                       // buffer de datos
char number[10];

int* paralel_read_score(int max_iteracions, int num_iter) {

   int status;
   int pid;
   int i;
   int array_temp[max_iteracions - num_iter];
   int* res;
   int p[2], readbytes;
   char buffer[1024];
   char* tok;

   pipe( p );

   /* GNU Chess 5.1_devesa Beta
    *
    * l'ultim dels fitxers que llegim     */

   if(max_iteracions == (num_iter + 1)) {
         array_score[num_iter] = read_score(num_iter);
         movim[num_iter]->score = array_score[num_iter];
         array_temp[0] = array_score[num_iter];
         res = array_temp;
         return res;
   }
   else {

         if((pid=fork()) == 0) {

               /* GNU Chess 5.1_devesa Beta - FILL !!             */
               res = paralel_read_score(max_iteracions, num_iter+1);

               /* GNU Chess 5.1_devesa Beta.                              */
               /* Escribim en el PIPE el valor de les variables   */
               close(p[0]);
               for(i=num_iter+1; i<max_iteracions; i++)
               {
                     array_score[i] = *res;
                     sprintf(buffer, "%d/", array_score[i]);
                     write( p[1], buffer, strlen( buffer ) );
                     res++;
               }
               close(p[1]);

             exit(1);

         }
         else {


              array_score[num_iter] = read_score(num_iter);
              movim[num_iter]->score = array_score[num_iter];
              waitpid(pid,&status,0);
```

```
                /* GNU Chess 5.1_devesa Beta.
*/
                /* Llegim del fill el que en dÃ³na a travÃ©s del pipe, */
                /* i convertim els bytes llegits en un token per a   */
                /* passar-lo a un array dels valors de tots els      */
                /* fills anteriors
*/
                 close( p[1] );
                   readbytes= read( p[0] , buffer, 1024);
                 close( p[0] );

                 i = num_iter+1;
                 tok=strtok(buffer,"/");
                 array_score[i] = atoi(tok);
                 i++;
                 for(i; i<max_iteracions; i++) {
                     tok = strtok( NULL, "/" );
                     array_score[i] = atoi(tok);
                 }

                if(num_iter != 1) {
                       res = &array_score[num_iter];
                       return res;
                }
                else {
                        for(i=0; i<max_iteracions; i++) {
                             printf("\t%d - %d\n",movim[i]->move,
                            array_score[i]);
                       }
                       printf("\n");
                       res = &array_score[1];
                       return res;
                }

          }

    }
}

/* GNU Chess 5.1_devesa Beta.                          */
/* FunciÃ³ encarregada de llegir del fitxer           */

int read_score(int num_iter) {

       int score;
       int i;
       char* tok;
       char atokenizar[100];
       char name_file[70] = "/home/jdevesa/files/search.param";
       sprintf(name_file, "%s_%d", name_file, num_iter);

       prm_f=fopen(name_file,"r+");

       while(fgetc(prm_f) == -1) {}
       if(verbose_condor == 1)
             printf("PFC:Leyendo el valor de las variables de
       'chess_out.param'!\n");
```

```
		/* PFC GNU Chess 5.1 Beta - llegint les variables de tipus enter
	*/
	if(verbose_condor == 1)
		printf("PFC: Leyendo enteros en el fichero de parametros....");

	fgets(buffer,200,prm_f);
	fgets(logStream,150,prm_f);

	strcpy(atokenizar,logStream);
	tok=strtok(atokenizar,"\t");
	for(i=0; i<5; i++) {
	   tok = strtok( NULL, "\t" );
	   if(i==0) score = atoi(tok);
	}
	load_logfile('P');
	// fprintf (prm_f,"%d\t\t%d\t%s\t%s\t%f\n", TreePtr[1]-
	>move,score,substr(timeIni,11,8),substr(asctime(tmPtrFin),11,8),etExe);
	fclose(prm_f);
	//remove(name_file);

	return score;
}
```

3.3.4. Avaluació dels moviments

Un cop rebudes les execucions i assignades correctament al *array* de moviments, queda pendent la avaluació del millor moviment i el seu corresponent valor numèric. Aquests dos valors es guarden en la variable global `int RootPV;` i en la variable local `int best`, que serà la variable que retornarem a la iteració principal.

En l'execució seqüencial, el calcul del valor del moviment i la seva avaluació es feien un just darrera l'altre. Si el valor no complia com a millor moviment, ens oblidàvem d'ell.

En aquest cas no es així. Hem recuperat tots els valors dels fitxers *search.param_N* sense prendre cap decisió i ara ens toca avaluar-los. Aquesta avaluació és força senzilla i s'executa dins d'un bucle for:

```
for(i = 0; i < num_moviments;  i ++) {

	if(movim[i]->score > best) {
```

```
                best = movim[i]->score;
                pbest = movim[i];

                if (best > alpha)
                {
                      rootscore = best;
                      RootPV = movim[i]->move;
                      if (best >= beta)
                            return best; //poda alpha-beta
                      ShowLine (RootPV, best, '&');
                }

          }

      ....
}

return best;
```

3.3.5. Fitxers de logs

Per fer un seguiment de les execucions i per avaluar els desenvolupaments, s'ha creat un sistema de *logs* que mantenen la informació de les partides, bàsicament les avaluacions del *Min-Max*, el temps de còmput i el nombre de nodes generats i avaluats de cada torn de la computadora.

Cada fitxer s'anomena *log_(num_pid)* segons el numero de tasca que *Linux* li dona al procés, així ens asseguràvem tenir pràcticament un identificador únic de fitxer i no haver de mantenir un registre de *logs* per tal de no sobreescriure cap fitxer.

Els fitxers tenen 3 tipus de registres:

- Registres Jugador: únicament escriuen a *Condor* el moviment que ha fet el jugador, indicant si juga amb blanques o amb negres.
- Registres Condor: ja hem dit anteriorment que el fitxer de sortida de les execucions remotes escriuen una sèrie de variables d'on, des d'un punt de vista algorísmic, només aprofitem el valor heurístic del calcul.

  Per una banda, agafarem el valor '*score*' , però per l'altre agafem tota la línia que llegim del *search.param_N* per a transcriure-la directament al fitxer de *log*, afegint l'*String Condor* a davant de tot per a distingir aquest registre.

La informació que escrivim en aquest registres és:

- moviment.

- valor.

- hora d'inici de l'execució.

- hora final de l'execució.

- durada de l'execució.

- nodes calculats.

- nodes generats.

Aquesta es una bona manera de tenir controlades totes les execucions de *Condor* i de poder tenir un control del temps d'execució i dels nodes més realista.

- Registres Computadora: Aquests registres són el resultat total del càlcul i escriuen la mateixa informació que els registres *Condor*, però amb el temps d'execució i els nodes de la pròpia màquina i el moviment i valor definitius de l'execució.

Si estem jugant amb *Condor*, tindrem un registre *Condor* per a cada execució remota i finalment un registre Computadora, si pel contrari juguem amb l'algorisme seqüencial inicial, només tindrem un registre Computadora per a cada jugada de la màquina.

Per fer els càlculs dels nodes totals en el banc de proves, serà suficient sumar tots els registres de nodes calculats i nodes generats dels registres *Condor* més el registre Computadora. En canvi, per calcular el temps de càlcul, com que estem fent una execució paral·lela, ens quedarem amb el màxim de tots els temps del registre *Condor* per obtenir el temps de execució. I ens quedarem amb el temps obtingut en el registre Computadora per obtenir el temps real (en el temps d'execució no tenim en compte el *delay* que aplica la configuració de *Condor*).

Apart d'aquests canvis, i per identificar les diferents execucions, hem creat una

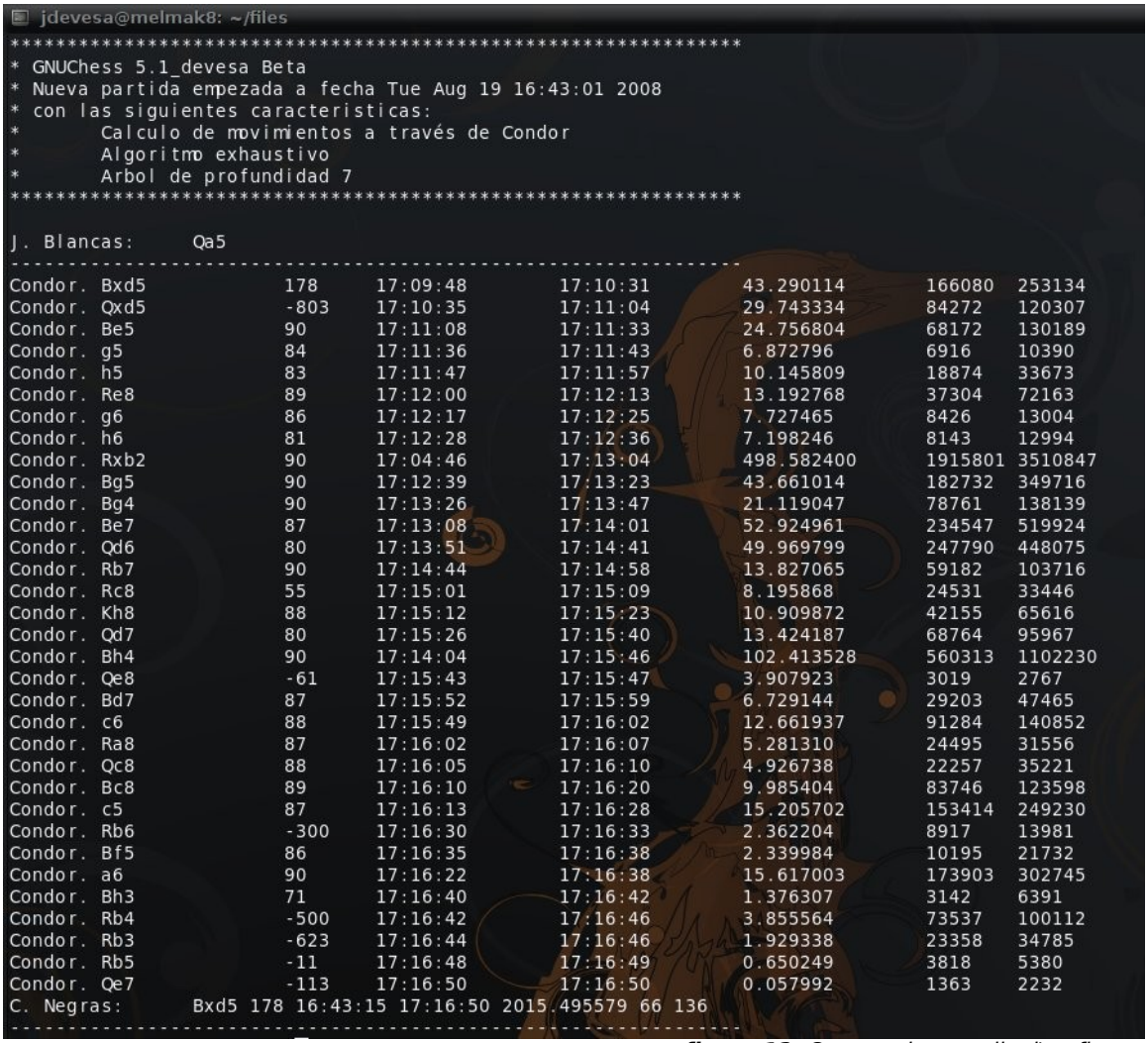
capçalera de fitxer on hi mostrem les característiques (profunditat, algorisme, mode de calcul) de cada partida.

Tot seguit mostro el codi del programa que escriu el fitxer de *logs* (tenir en compte que *logstream* es una variable global)

```
/* Creem un fitxer de log per controlar el temps de les partides  */
void load_logfile(char jugador) {

	time_t now;
	struct tm *tmPtrIni;

	if((log_f=fopen(nom_fitxer_log,"a+"))!=NULL) {
		if(jugador == 'N') {

			/* Establim el moment que comenÃ§a la partida */
			now = time(NULL);
			tmPtrIni = gmtime(&now);
				fprintf(log_f,"****************************\n");
			fprintf(log_f,"* GNUChess 5.1_devesa Beta\n");
			fprintf(log_f,"* Nueva partida empezada a fecha %s",
				asctime(tmPtrIni));
			fprintf(log_f,"* con las siguientes caracteristicas: \n");
		    if(condor == 1) fprintf(log_f,"*\tCalculo de movimientos a
		                travÃ©s de Condor\n");
		    else fprintf(log_f,"*\tCalculo de movimientos en la propia
			maquina\n");
		    if(algorithm == 1) fprintf(log_f,"*\tAlgoritmo iterativo\n");
		    else if(algorithm == 2) fprintf(log_f,"*\tAlgoritmo
				exhaustivo\n");
		    fprintf(log_f,"*\tArbol de profundidad %d\n", SearchDepth);
			fprintf(log_f,"*************************\n\n");

			}

		if(jugador == 'J') {
			if(board.side == 1) fprintf( log_f,"J. Blancas:\t%s\n",
			Game[GameCnt].SANmv);
			else fprintf(log_f,"J. Negras:\t%s\n", Game[GameCnt].SANmv);
				fprintf(log_f,"----------------------\n");
		}

		if(jugador == 'C') {
			if(board.side == 1) fprintf( log_f,"C. Blancas:\t%s\n",
			logStream);
			else fprintf(log_f,"C. Negras:\t%s\n", logStream);
				fprintf(log_f,"----------\n");
		}
```

```
            if(jugador == 'P') {
                  fprintf( log_f,"Condor. %s\n", logStream);
            }
      }
      else printf("Unable to open log file!\n");
      fclose(log_f);
}
```

i una captura de pantalla del fitxer de *logs* que correspon a una execució mitjançant *Condor* de l'exemple de l'apartat 2.2.3.4 (rendiment de l'algorisme *GNU Chess*), que es pot apreciar a la figura 12, i on es pot observar com el jugador (J) fa el moviment Qa5 i l'ordinador (C) el contesta amb un Bxd5. Els registres que comencen per *Condor* representen cada un dels càlculs de l'execució remota.



```
jdevesa@melmak8: ~/files
****************************************************************
* GNUChess 5.1_devesa Beta
* Nueva partida empezada a fecha Tue Aug 19 16:43:01 2008
* con las siguientes caracteristicas:
*       Calculo de movimientos a través de Condor
*       Algoritmo exhaustivo
*       Arbol de profundidad 7
****************************************************************

J. Blancas:     Qa5
----------------------------------------------------------------
Condor. Bxd5            178     17:09:48        17:10:31        43.290114       166080  253134
Condor. Qxd5            -803    17:10:35        17:11:04        29.743334       84272   120307
Condor. Be5             90      17:11:08        17:11:33        24.756804       68172   130189
Condor. g5              84      17:11:36        17:11:43        6.872796        6916    10390
Condor. h5              83      17:11:47        17:11:57        10.145809       18874   33673
Condor. Re8             89      17:12:00        17:12:13        13.192768       37304   72163
Condor. g6              86      17:12:17        17:12:25        7.727465        8426    13004
Condor. h6              81      17:12:28        17:12:36        7.198246        8143    12994
Condor. Rxb2            90      17:04:46        17:13:04        498.582400      1915801 3510847
Condor. Bg5             90      17:12:39        17:13:23        43.661014       182732  349716
Condor. Bg4             90      17:13:26        17:13:47        21.119047       78761   138139
Condor. Be7             87      17:13:08        17:14:01        52.924961       234547  519924
Condor. Qd6             80      17:13:51        17:14:41        49.969799       247790  448075
Condor. Rb7             90      17:14:44        17:14:58        13.827065       59182   103716
Condor. Rc8             55      17:15:01        17:15:09        8.195868        24531   33446
Condor. Kh8             88      17:15:12        17:15:23        10.909872       42155   65616
Condor. Qd7             80      17:15:26        17:15:40        13.424187       68764   95967
Condor. Bh4             90      17:14:04        17:15:46        102.413528      560313  1102230
Condor. Qe8             -61     17:15:43        17:15:47        3.907923        3019    2767
Condor. Bd7             87      17:15:52        17:15:59        6.729144        29203   47465
Condor. c6              88      17:15:49        17:16:02        12.661937       91284   140852
Condor. Ra8             87      17:16:02        17:16:07        5.281310        24495   31556
Condor. Qc8             88      17:16:05        17:16:10        4.926738        22257   35221
Condor. Bc8             89      17:16:10        17:16:20        9.985404        83746   123598
Condor. c5              87      17:16:13        17:16:28        15.205702       153414  249230
Condor. Rb6             -300    17:16:30        17:16:33        2.362204        8917    13981
Condor. Bf5             86      17:16:35        17:16:38        2.339984        10195   21732
Condor. a6              90      17:16:22        17:16:38        15.617003       173903  302745
Condor. Bh3             71      17:16:40        17:16:42        1.376307        3142    6391
Condor. Rb4             -500    17:16:42        17:16:46        3.855564        73537   100112
Condor. Rb3             -623    17:16:44        17:16:46        1.929338        23358   34785
Condor. Rb5             -11     17:16:48        17:16:49        0.650249        3818    5380
Condor. Qe7             -113    17:16:50        17:16:50        0.057992        1363    2232
C. Negras:      Bxd5 178 16:43:15 17:16:50 2015.495579 66 136
----------------------------------------------------------------
```

figura 12: *Captura de pantalla d'un fitxer de logs*

3.3.6. Opcions inicials.

Una de les primeres funcions que es criden en *GNU Chess* abans de realitzar cap moviment és la funció *Initialize()*, que inicialitza totes les variables i estructures i deixa a punt el joc per començar la partida.

A mode de complement i independent del canvi que hem fet fins ara per convertir el joc seqüencial en un joc paral·lel, hem trobat interessant modificar aquesta funció per a poder escollir d'una manera interactiva les següents variables:

- Algorisme: es demana que l'usuari esculli entre un algorisme Iteratiu (el GNU Chess original, PVS + *Iterative Deepening*), un algorisme Exhaustiu (el que acabem d'explicar) i un algorisme Mixt (el que explicarem en el següent apartat)
- Mode de computació: Si volem que l'algorisme es calculi a través de *Condor* (en paral·lel) o bé si volem que l'algorisme es calculi en la pròpia màquina.
- Profunditat: En el *GNU Chess* original tenim l'opció de posar la comanda **depth n** on establim la profunditat de l'algorisme. Si no posem això , el programa *GNU Chess* decideix un temps màxim de càlcul per a cada jugada, arribi on arribi l'arbre iteratiu.

  A nosaltres ens interessa fer un estudi sobre el comportament de diferents algorismes i treure uns rendiments segons la profunditat, es per això que forcem que la variable de profunditat es defineixi de bon principi

Les combinacions que realment ens interessen per les proves són:

- Iteratiu + computació normal.
- exhaustiu + *Condor*.
- mixt + *Condor*.

Tot i així deixem a escollir qualsevol combinació (menys mixt + normal, que no té

sentit, com ja veurem a continuació). per si necessitem fer alguna prova o per deixar per demostrar que algunes combinacions (exhaustiu + normal) ja ha quedat més que demostrat que tenen poca eficiència.

Com en l'apartat anterior, deixem el codi creat i una captura de pantalla.

```
void Initialize ()
/**************************************************************************
 *
 *  The main initialization driver.
 *
 **************************************************************************/
{
   InitLzArray ();
   InitBitPosArray ();
   InitMoveArray ();
   InitRay ();
   InitFromToRay ();
   InitRankFileBit ();
   InitPassedPawnMask ();
   InitIsolaniMask ();
   InitSquarePawnMask ();
   InitBitCount ();
   InitRotAtak ();
   InitRandomMasks ();

   InitDistance ();
   InitVars ();
   InitHashCode ();
   InitHashTable ();
   CalcHashKey ();

   /* PFC GnuChess 5.1_devesa Beta                                    */
   /* S'inicialitza per defecte el sistema condor   */
   InitCondor();

#ifdef HAVE_LIBREADLINE
   using_history();
#endif
}


void InitCondor ()
{

       char tipo_comp[1];
       int profundidad;
       char alg[1] ;

       while(true)
       {
                     printf("Introduce el algoritmo para recorrer el arbol,
(I)terativo/(E)xhaustivo/(M)ixto: ");
                     alg[0] = getchar();
                     if (alg[0] == 'I') { algorithm = 1; getchar(); break;}
                     if (alg[0] == 'E') { algorithm = 2; getchar(); break;}
                     if (alg[0] == 'M')
                     {
                            algorithm = 3;
                            condor=0;
                            numSecLev=0;
                            getchar();
```

```
                                break;
                        }
        }
        /* si el algoritmo es mixto utilizaremos siempre unas iteraciones sin condor i unas con
Condor,
         * por esto es inutil preguntar
                                                                        */
        if(algorithm != 3) {
                while(true)
                {
                        printf("Introduce el modo de computacion del juego , (C)ondor/(N)ormal:
");
                        tipo_comp[0] = getchar();
                        if (tipo_comp[0] == 'C') { condor = 1; getchar(); break;}
                        if (tipo_comp[0] == 'N') { condor = 0; getchar(); break;}
                }
        }
        printf("Introduce la profundidad del arbol (1-20): ");
        scanf("%d", &profundidad);
        if(profundidad> 0 && profundidad <21) {
                SearchDepth = profundidad;
        }
        else {
                SearchDepth = 7;
                printf("Eleccion incorrecta. Se usa un arbol de profundidad 7 por defecto\n");
        }
        //Inicializamos el fitxero de log de la partida
        sprintf(nom_fitxer_log,"%s_%d",nom_fitxer_log, getpid());
        load_logfile('N');
        verbose_condor = 0;
        condor_monitor = 0;


}
```

i la captura de pantalla:

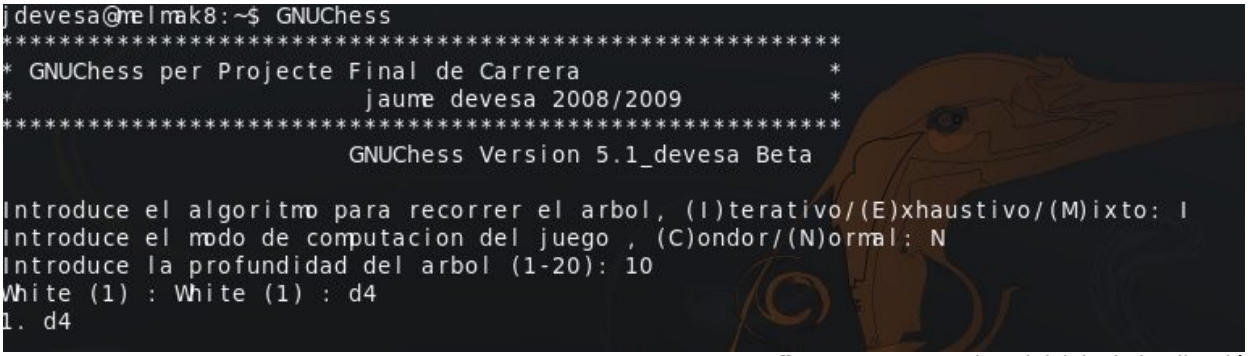


figura 12+1: *Opcions inicials de l'aplicació*

4. GNU Chess Mixt. Canvis realitzats.

Aquest es un apartat 'germà' de l'anterior, i com a tal, segueix el mateix procés. Primer de tot farem una reflexió del que volem aconseguir, després, a través de *pseudocodi* i gràfics plasmarem una idea esquemàtica de l'evolutiu, i finament, en el tercer apartat entrarem en els canvis propis d'aquest desenvolupament.

4.1 Reflexió sobre la computació mixta.

Un cop aconseguit l'objectiu de 'tallar' l'aplicació *GNU Chess* i adaptar-la perquè s'executi en *Condor,* i essent conscients dels seus defectes de “intel·ligència”, era pràcticament obligatori, servint-nos dels coneixements adquirits en l'estudi del PVS, proposar un algorisme propi. En aquest apartat veurem el que he anomenat el *GNU Chess* Mixt. He intentat agafar els aspectes que més m'han convençut dels dos paradigmes de càlcul per fer una execució que combina el PVS amb el càlcul paral·lel. Més endavant veurem si hem tingut èxit.

Un dels problemes més significatius del *Min-Max* paral·lel que hem desenvolupat anteriorment és la falta d'informació sobre *alpha-beta*. Es calculen masses nodes que, si executéssim el PVS iteratiu no es calcularien. Això es deu a que l'arbre es desplega una sola vegada i desordenadament, perdent molta eficiència de càlcul, tal i com ja hem demostrat en l'apartat del estudi sobre el rendiment del PVS.

El que ens porta a preguntar-nos: no guanyaríem en intel·ligència si l'arbre estigués ordenat i només enviéssim a *Condor* els n primers fills del node arrel i la resta els

calculéssim a la aplicació original? No evitaríem, a més d'això, que la xarxa i la cua d'execucions es col·lapsessin d'enviaments que un cop a l'ordinador remot simplement aplicarien la poda?

Aquesta es la ideologia en que es basa aquesta nova evolució i el que hem volgut aconseguir. Ara bé... per fer això hem d'aplicar un *Iterative Deepening* que ordeni l'arbre prèviament i que no suposi una carrega massa gran de temps per a continuar considerant aquesta evolució com a 'paral·lela'. Fins a quin nivell, doncs, s'ha d'ordenar per adquirir aquests requisits?

Les proves que anteriorment hem esmentat sobre diferents punts de la partida i que més endavant veure'm ens serviran per això. Sobre les diferents mostres de cada partida hem creat un gràfic que ensenya la mitjana de temps que necessita Condor per a calcular una jugada per a cada una de les profunditats. Aquest temps, evidentment, incrementa exponencialment per a cada nivell de profunditat major. Aquest és el gràfic:

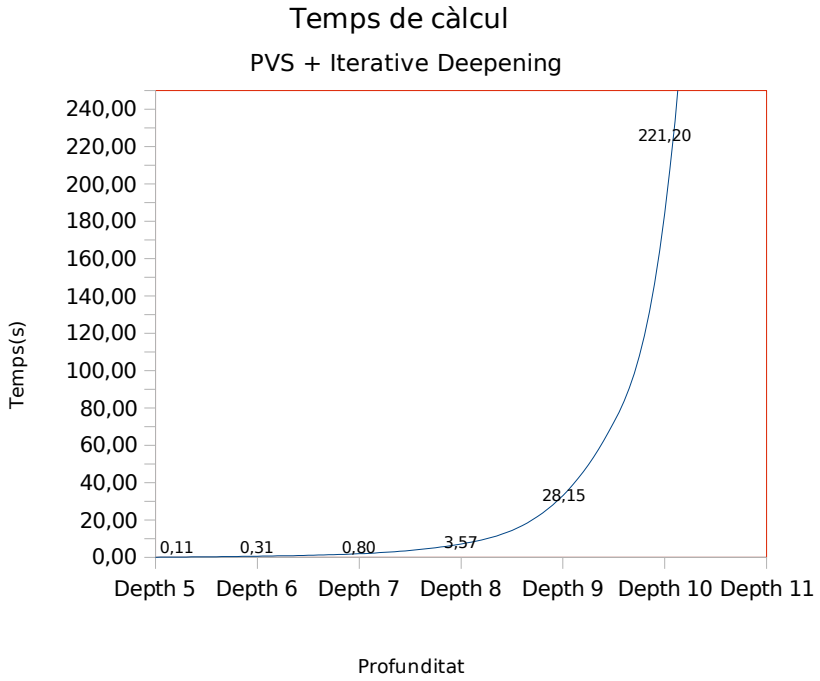


Figura 14: *Mitjana de temps segons profunditat del PVS+Iterative Deepening*

El valor de la profunditat 11 es 1570.25 segons, però el rang se'ns anava massa lluny

i ja queda prou clar el caràcter exponencial de la funció.

Suposant doncs, que a cada nivell que anem pujant ordenem millor l'arbre però no volem perdre massa temps en l'ordenació, s'ha decidit ordenar l'arbre fins a profunditat 7 (0.8 segons de mitjana) i llavors llençar la aplicació paral·lela.

Hem "resolt" una part del problema, però ens queda una altra part: Quants fills creiem necessaris a enviar perquè l'enviament estigui justificat? Encara que el nombre de fills sigui 7 o 8... això no es desaprofitar molt *Condor*?

Per tant, estem en un conflicte d'interessos: hem d'enviar suficients fills a *Condor* per aprofitar la seva potència de càlcul, però a la vegada, no li volem enviar treballs 'inútils' per no sobrecarregar la xarxa ni la cua. I aquí tenim la principal modificació en respecte a l'anterior evolució: **No enviarem a *Condor* els fills del node arrel, sinó els fills dels seus fills.** Així tindrem una cua de fills 'dignes' de calcular i al mateix temps aprofitem les avantatges que ens dona *Condor*.

La idea és: obtenir la quantitat de maquines lliures de *Condor* (a través de la comanda *condor_status*, comentada l'apartat 2.1.4) i fer una divisió entera d'aquest nombre per 30 (*branching factor* dels jocs d'escacs) i li sumarem 1 (si tenim 25 maquines, per exemple, el resultat serà 0+1=1). Aquesta serà la manera de calcular el nombre de nodes fill del *root* als quals aplicarem el càlcul en paral·lel, la resta es calcularà a la màquina pròpia.

Exemplifiquem-ho:

Suposem que es el torn de l'ordinador que té profunditat 10. Els seus passos seràn:

1. Calcular els primers 7 nivells de profunditat, amb PVS i ordenant l'arbre.
2. Desplegar l'arbre fins a profunditat 10, segons l'ordre que ens ha deixat el punt anterior.
3. Obtenir el nombre de màquines treballadores lliures de *Condor*. (Suposem unes 50)

4. Fem el càlcul: (50 div 30) + 1 = **2.** Els dos primers fills del node arrel enviaran els seus fills a través de *Condor*. La resta es calcularan en local.

D'aquesta manera creiem que aconseguirem un compromís entre la velocitat de càlcul de *Condor* i l'optimització de recorregut de l'arbre que ens dona el PVS.

4.2 Esquema gràfic i algorítmic de la nova aplicació.

Com en l'anterior cas, intentarem crear un pseudocodi que exemplifiqui el que hem desenvolupat i explicat en el punt 4.1 i partint de la base del pseudocodi inicial . Comencem am la funció iterativa:

```
function iterateMixt(prof_max) is
      alpha = -INF;
      beta = INF;
      int prof_act = 1;
      mentre prof_act <= 7
            pvsRoot(node_root, prof_act, alpha, beta);
            actualitza_alpha_beta();
            prof_act++;
     fi_mentre;
      pvsRootMixte(node_root, prof_max, alpha, beta);
      actualitza_alpha_beta();
     return node_root->moviment;
fi_function
```

Veiem com ja tenim canvis en la funció. Fem una iteració fins a profunditat 7 per a ordenar l'arbre, i just després cridem a un pvsRootMixt amb la profunditat màxima, que s'encarregarà de distribuir quins nodes es calculen a través de *Condor*, i quins es calculen en local.

```
function pvsRootMixte(node, profunditat, alpha, beta) is

      num_fillsCondor = calculaFillsCondor()
      num_fillsRoot = calculaFillsRoot()
      for num_fillsCondor to num_fillsRoot
            fill = seguentFill(node)
            si nou_thread()== fill llavors
                  pvsMixt(fill, profunditat, alpha, beta);
            fi_si
      fi_for
      for num_fillsCondor to num_fillsRoot

            fill = seguentFill(node);
            valor = pvs(fill, profunditat, alpha, beta);
            si (valor > millor) llavors
                  millor = valor;
                  node_root->moviment = fill->moviment;
            fi_si

      fi_for

fi_function
```

Aquesta funció (*pvsRootMixte*) estableix una serie de fills (*num_fillsCondor*) que s'obtindran a través de la comanda *condor_status* i que els seus respectius fills es calcularan a traves de *Condor*. Es important detectar que aquesta crida es fa a través de *threads*, per a què totes les crides es facin simultàniament. La resta de fills es calcularan amb un algorisme PVS estàndard, però aprofitant l'acotament de *alpha-beta* que ens proporcionaran la primera tanda de fills. La idea es que la majoria d'aquests fills calculats iterativament podin les seves branques i la feina 'bruta' es faci en paral·lel.

I per acabar ficarem el pseudocodi de la funció pvsMixt que farà el mateix que el *pvsRootParalela* del punt 3 però en un nivell de profunditat inferior:

```
function pvsMixt(node, profunditat, alpha, beta) is

     mentre teMesFills(node)
            fill = seguentFill(node);
            preparaFitxerCondor(fill, profunditat, alpha, beta);
      fi_mentre
      cridaCondor();
      tornaIniciNode(node);
      mentre teMesFills(node)
            si creem_Thread = fill llavors
                  node->valor = esperaExecucióCondor()
            fi_si
            sinó
                  continua();
            fi_sinó
      fi_mentre
      tornaIniciNode(node)
      mentre teMesFills(node)
            si (node->valor > millor) llavors
                  millor = node->valor;
                  node_root->moviment = node->moviment;
      fi_mentre
fi_function
```

El que ens porta a una execució a l'aplicació *ParalelSearch* que serà la mateixa que en el punt 3.

Des d'un punt de vista esquemàtic tenim el següent:

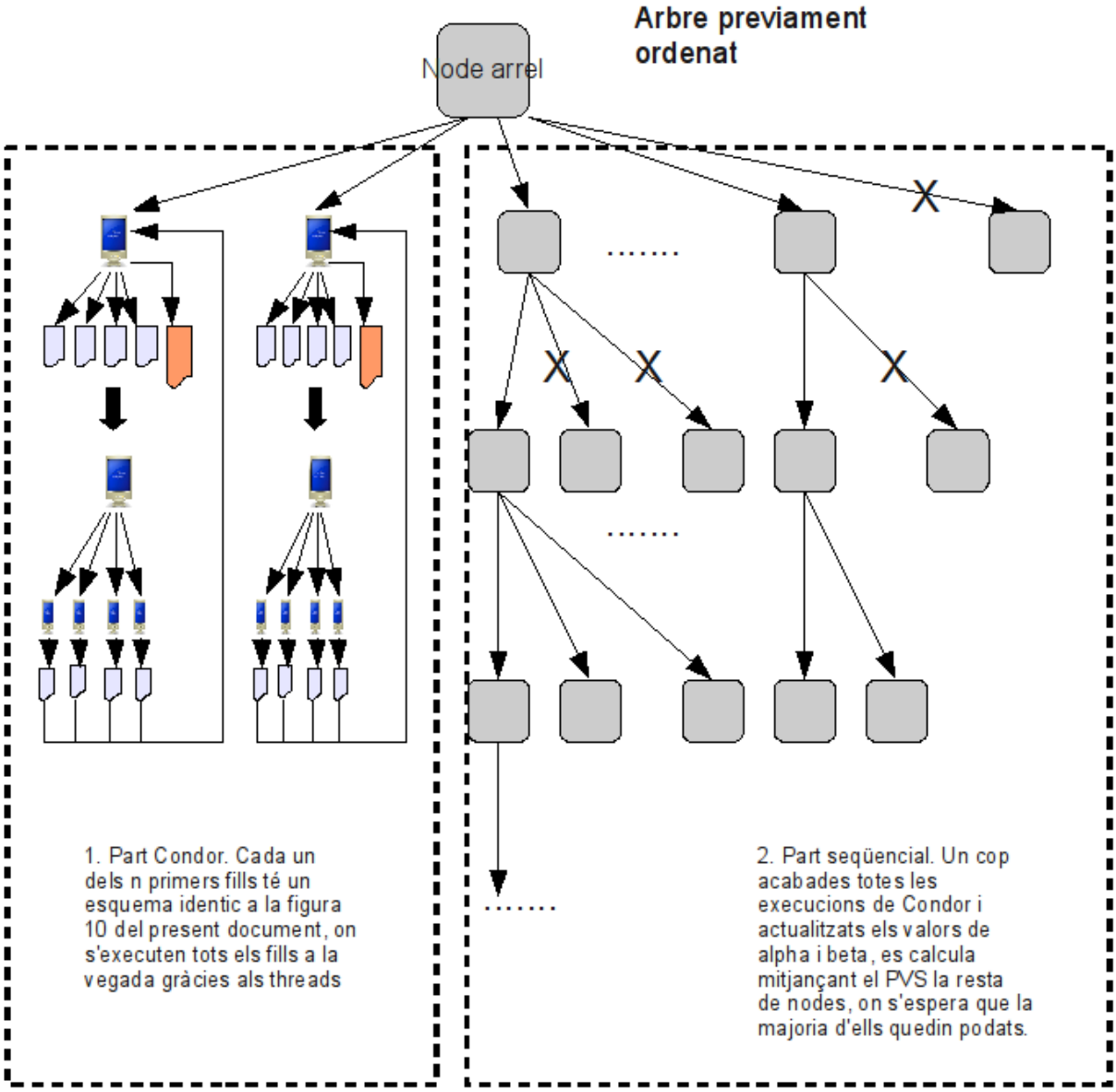


***Figura 15**: esquema de l'execucio Mixta*

4.3 Detall dels canvis realitzats.

En aquest cas, la crida a *Condor*, la forma de recuperar els resultats, el sistema de *logs* o el menú inicial són exactament iguals que en la crida en paral·lel. Els canvis, més que noves funcionalitats o una manera de calcular les coses de manera diferent, són canvis estructurals, on les mateixes funcions es criden d'es de altres llocs. Anem

a veure'ls:

4.3.1. Iteració principal.

La fi de la iteració principal en *GNU Chess* va definida per dues variables, la variable *Idepth* i la variable TIMEOUT, segons si definim una profunditat màxima o no. Com que forcem la definició de la profunditat màxima al menú que hem creat, sempre sortirem del bucle principal quan *Idepth* arribi al seu màxim.

Desconeixem perquè, però la condició de sortida és *Idepth == SearchDepth * DEPTH* on *SearchDepth* es el valor de profunditat que hem definit en el menú principal com a màxim i DEPTH es una constant que val 12. Per tant, si per exemple, definim una profunditat de 10, per forçar la sortida hem de igualar el *Idepth* a 120. Es una mica confús que si volem un arbre de profunditat 1 hem de passar-li al procés *SearchRoot* un valor de 12, si volem un arbre de profunditat 2 un valor 24, etc... Resulta confús, però és així.

Nosaltres ens hem adaptat a la característica i hem modificat el codi de la següent manera:

```
Idepth += DEPTH;

/* Quan hem fet 6 iteracions creiem que ja tenim l'arbre prou ordenat     */
/* i cridem a condor perque s'encarregui dels sub-arbres                  */
if(Idepth == DEPTH*8) {
        condor = 1;
        Idepth = SearchDepth * DEPTH;
}
/* PFC GNUChess 5.1_devesa Beta
/* Segons el tipus de computacio triarem un procediment o un    */
/* altre */
if(condor == 0) score = SearchRoot (Idepth, RootAlpha, RootBeta);
else score = SearchRootMixt(Idepth,RootAlpha,RootBeta,num_maquines);
```

La primera ordre augmenta en 1 (en 12 en realitat) la profunditat de l'algorisme. El següent **if** comprova si hem passat ja de la setena iteració i ens trobem ja en la 8ena. Si es així assignem a la variable *condor* el numero 1, que significa que el pròxim càlcul es farà a través de *Condor*; i a la variable *Idepth* la profunditat màxima.

La següent condició estableix la diferencia entre si encara estem en les 7 iteracions inicials que ordenen l'arbre o si ja estem en el desplegament final. I per a cada cas cridem a una rutina o a una altra.

Més endavant ens trobarem amb la condició:

```
if (Idepth == SearchDepth*DEPTH)
   break;
}
```

que ens permetrà sortir del bucle principal.

4.3.2. Comprovació de les màquines disponibles.

Ja hem dit que el nombre de fills que enviarem a *Condor* dependrà de les maquines disponibles a *Condor*, per aprofitar al màxim la seva capacitat de càlcul. Aquest càlcul el farem a la iteració principal, cridant a una subrutina que assignarà un valor a la variable **num_maquines** (Veure que en el codi de l'apartat 4.3.1 cridem a la funció **SearchRootMixt** amb aquesta variable).

Per saber el nombre de maquines disponibles, es suficient fer una crida a *condor_status -avail* des de consola. Aquest és el resultat:

```
jdevesa@melmak8:~$ condor_status -avail

Name          OpSys       Arch   State      Activity   LoadAv Mem   ActvtyTime

vm1@melmak8.0 LINUX       INTEL  Unclaimed  Idle       1.590  1645  0+00:00:04
vm2@melmak8.0 LINUX       INTEL  Unclaimed  Idle       29.610  1645  0+00:00:05

                     Total Owner Claimed Unclaimed Matched Preempting Backfill

        INTEL/LINUX      2     0       0         2       0          0        0

               Total     2     0       0         2       0          0        0
jdevesa@melmak8:~$
```

***Figura 16**: Crida al condor_status per saber les màquines disponibles*

En aquest cas en surten 2 perquè estem executant en local i crec que *Condor* distingeix entre la maquina **localhost** i la màquina **127.0.0.1**

Per formatar aquest resultat, d'on únicament n'aprofitem el numero de màquines, farem una crida mitjançant *awk:*

```
condor_status -avail | grep INTEL/LINUX | awk '{ print $5 }'
```

Figura 17: *sortida de la comanda condor_status formatejada*

Per fer això des de codi utilitzarem la comanda **system** i a més redirigirem la sortida a un fitxer anomenat *status* que posteriorment llegirem. Al llegir aquest fitxer assignarem directament el valor que llegim a la variable que retornem. Aquí tenim el codi de la rutina:

```
/* PFC GNUChess 5.1_devesa Beta
                      */
/* FunciÃ³n que genera un fichero con una llamada al sistema que contiene */
/* el numero de maquinas disponibles en condor
      */
int availMachines(void) {

       FILE *sta_f;
       int num_machines;
       system("condor_status -avail | grep INTEL/LINUX | awk '{ print $5 }' >
/home/jdevesa/files/status");
       if((sta_f=fopen("/home/jdevesa/files/status","a+"))!=NULL) {
              fscanf(sta_f,"%d",&num_machines);
              fclose(sta_f);
              return num_machines;
       }
       else {
              printf("Unable to open status file");
              return 0;
       }

}
```

4.3.3. Modificació del procediment pare de la cerca.

Comportament de la part paral·lela.

En l'apartat 3 tots els fills del node principal (anomenem-los de primer ordre) s'invoquen al mateix temps a través de *Condor*. En aquest cas no podem fer això. Hem de diferenciar entre els fills de primer ordre que enviaran els seus fills (anomenem-los de segon ordre) a *Condor* i els que s'executaran seqüencialment a la pròpia màquina.

Els fills de segon ordre que s'executaran a *Condor* s'han d'invocar de manera que els seus pares (fills de primer ordre) pràcticament els cridin al mateix temps. Només així tindrem una execució paral·lela i ho solucionarem a través de *threads*. El codi que explicarem a continuació forma part del pseudocodi anterior:

```
....
for num_fillsCondor to num_fillsRoot
      fill = seguentFill(node)
      si nou_thread()== fill llavors
            pvsMixt(fill, profunditat, alpha, beta);
      fi_si
fi_for
.....
```

L'estratègia de *threads* que hem fet servir en aquest cas es diferent de la explicada al punt 3.3.3. Abans teníem una estructura jeràrquica on cada *thread* era pare i fill a la vegada i llegia les dades del node anterior i, afegint les seves, les passava a un node superior. Aquest cop hem escollit fer-ho de manera que el procediment principal és l'encarregat de fer tots els threads i llegir el resultat de l'execució de cada un.

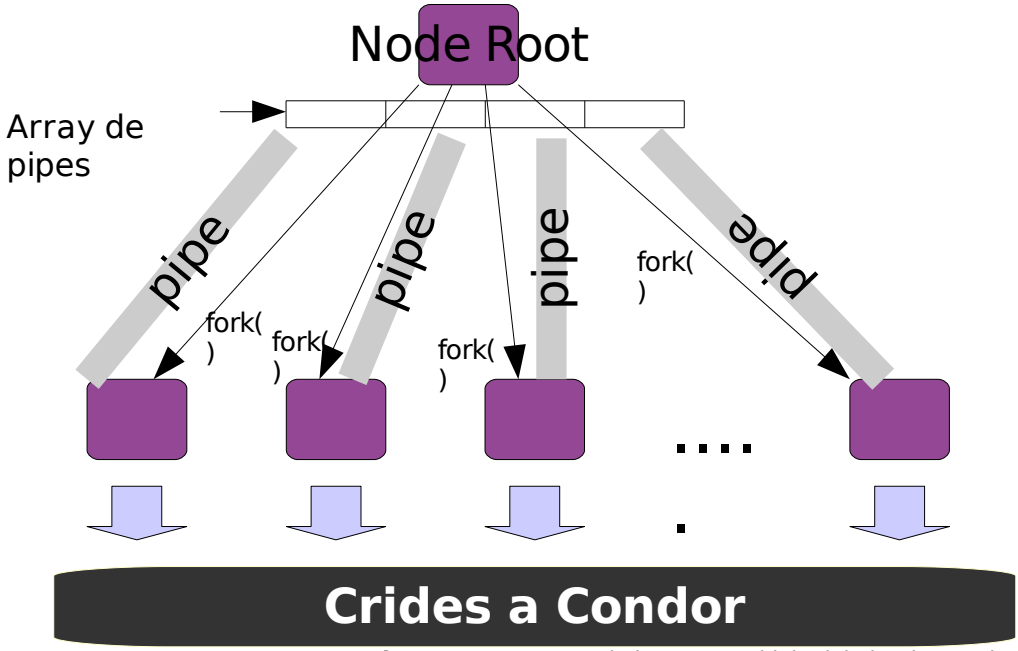


***Figura 16**: esquema de la part paral·lela del algorisme mixt*

El pare es comporta més o menys com en l'esquema anterior. Un cop acabades totes les execucions, s'actualitzaran els valors de *alpha* i *beta* i es cridarà la resta de fills de primer ordre, on s'esperen podes massives.

El codi corresponent a aquesta modificació és el següent:

```
/* GNU Chess 5.1_devesa Beta
                */

/* Amb els n primers moviments, que en teoria son els que es desplegaran  */
/* cridem al SearchMixt, que fara la crida a cÃ³ndor
        */
for(i=0; i<num_nodes_mixtes; i++) {


                if(i==0) movimRoot[i] = TreePtr[1];
                else {
                        p = movimRoot[i-1];
                        p++;
                        movimRoot[i] = p;
                }

                //aquesta part agafa el millor moviment que tenim de l'anterior operaciÃ³.
                //Correspon a la ordenaciÃ³ dinÃ mica
                pick (movimRoot[i], 1);

                pipe( pipe_act[i] );

                if((pid=fork()) == 0) {
```

```
                        ShowThinkingPar (movimRoot[i], ply);
                        MakeMove (side, &movimRoot[i]->move);
                        NodeCnt++;

                        if (movimRoot[i] == TreePtr[1])
                        {
                                score = -SearchMixt (2, depth-DEPTH, -beta,
                -alpha, nodetype, i);

                                if (beta == INFINITY && score <= alpha)
                                {
                                        score = -SearchMixt (2, depth-DEPTH,
                                -beta, -alpha, nodetype, i);
                                }
                        }
                        else
                        {
                                nodetype = PV;
                                alpha = MAX (best, alpha);
                                score = -SearchMixt (2, depth-DEPTH, -beta,
                -alpha, nodetype, i);
                        }

                        UnmakeMove (xside, &movimRoot[i]->move);
                        ply1score = movimRoot[i]->score = score;



                   /* GNU Chess 5.1_devesa Beta.                                   */
                   /* Escribim en el PIPE el valor de les variables */
                   close(pipe_act[i][0]);
                   sprintf(buffer, "%d", score);
                   printf("Escribim aixÃ² en el pipe %s\n",buffer);
                   write( pipe_act[i][1], buffer, strlen( buffer ) );
                   close( pipe_act[i][1] );

                exit(1);
                }

}

 /* GNU Chess 5.1_devesa Beta.                                              */
 /* Llegim del fill el que en dÃ³na a travÃ©s del pipe, */
 /* i convertim els bytes llegits en un token per a   */
 /* passar-lo a un array dels valors de tots els      */
 /* fills anteriors                                                          */

for(i=0; i<num_nodes_mixtes; i++) {

        //llegim el resultat del fill i
        close( pipe_act[i][1] );
        readbytes= read( pipe_act[i][0] , buffer, 1024);
        close( pipe_act[i][0] );

        //ho assignem al valor
        movimRoot[i]->score = atoi(buffer);


        //actualitzem tots els valors en cas que el fill que hem llegit
        //sigui el millor fins al moment.
        if (movimRoot[i]->score > best)
        {
                best = movimRoot[i]->score;
                pbest = movimRoot[i];

                if (best > alpha)
                {
                        rootscore = best;
                        RootPV = movimRoot[i]->move;
                        if (best >= beta)
                                goto done;
                        ShowLine (RootPV, best, '&');
```

```
                }
        }

        if (MATE+1 == best+1)
                return (best);

}

p = movimRoot[num_nodes_mixtes-1];

/* GNUChess 5.1_devesa Beta
        */
/* Amb els n segÃ¼ents, farem les crides iteratives normals                    */
/* , en prinipi, la majoria d'aquests moviments s'haurien de podar*/
for(i=num_nodes_mixtes; i<num_moviments; i++) {
```

4.3.4. Evolució dels fitxers de paràmetres.

El punt anterior condiciona un petit problema: el seu nom pot confondre la recepció i enviament dels fitxers. Ja hem dit en el seu moment que el nom dels fitxers es diferència per un sufix incremental: el primer fill tindrà el fitxer d'entrada *chess.param_1,* el segon fill *chess.param_2,* etc... Aquesta era una característica que ens anava molt bé per sincronitzar la sortida i arribada de fitxers i associar cada valor paral·lel. Ara ens suposa un problema:

Quan tenim més d'un *thread* que s'encarrega d'escriure i llegir fitxers i aquests tenen el mateix nom, el que aconseguim es que els seus fitxers de paràmetres es sobreescriguin i no puguem diferenciar entre el fill 1 del primer *thread* i el fill 1 del segon *thread*. El que ens ha portat a modificar la crida a les funcions *load_paramfile*, *appCondor* i *readscorefile.* Aquests tres procediments, que són d'escriptura, enviament i lectura dels fitxers de paràmetres, reben una variable addicional.

Si aquesta variable és -1, deixen el nom del fitxer tal i com està, per el contrari, si reben un valor numèric entre 0 i n, afegeixen un nou sufix al nom del fitxer. Aquest nou sufix correspondrà al numero de *thread* que l'està cridant.

D'aquesta manera, si estem executant l'algorisme exhaustiu enviarem a aquestes funcions la variable -1 i tot seguirà funcionant com abans. Si estem executant

l'algorisme mixt adaptarem el nom del fitxer de paràmetres al *thread* en qüestió i així podrem diferenciar els fitxers.

En aquest cas la modificació del codi es trivial.

5. Proves.

Aquest és l'apartat de proves. Aquí posarem a examen les modificacions que hem fet al *GNU Chess*, comprovarem els resultats i veurem si les suposicions que hem fet sobre cada algorisme (PVS+*Iterative Deepening*, Exhaustiu i Mixt) son correctes.

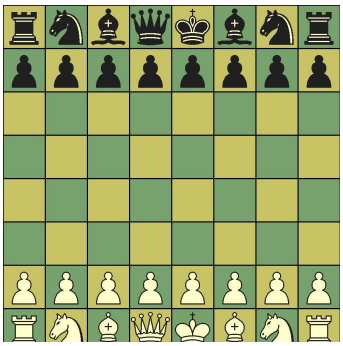
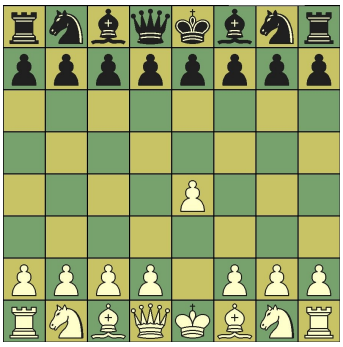
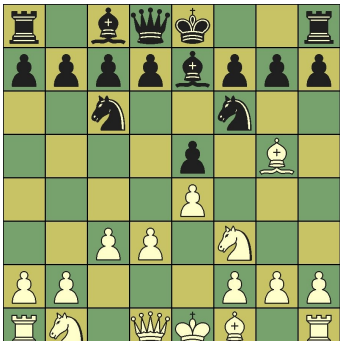
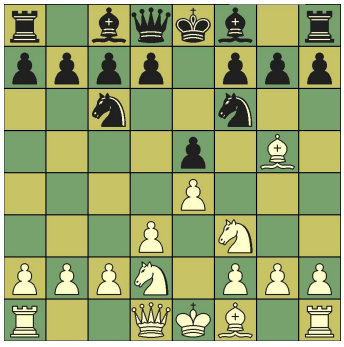
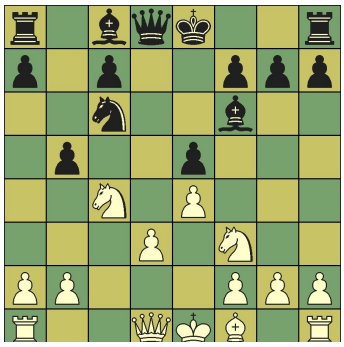
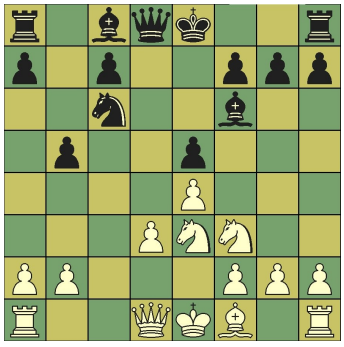
En el primer apartat mostrarem el banc de proves, una mica limitat però creiem que suficient per posar de relleu les avantatges i inconvenients de cada tipus d'execució. Cada un dels apartats següents contindrà gràfics i comentaris de cada un dels algorismes executats sobre el banc de proves. Per últim mostrarem els resultats finals comparant els tres algorismes i traient les conclusions que considerem oportunes.

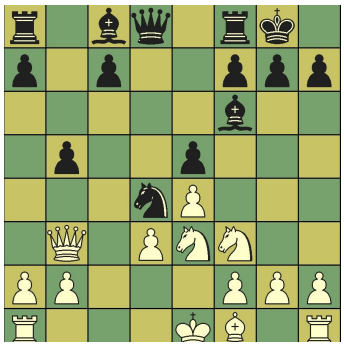
5.1 Banc de proves.

Una planificació ajustada del projecte i unes circumstancies adverses en el moment d'executar les proves van fer que la part dedicada al banc de proves es veiés molt limitada. El que volíem que fos un banc de proves de totes les mostres d'una partida per poder fer un seguiment molt ajustat de tots els algorismes es va retallar a unes mostres molt concretes sobre la partida, intentant copsar tots els punts clau per donar un resultat, el més fiable possible.

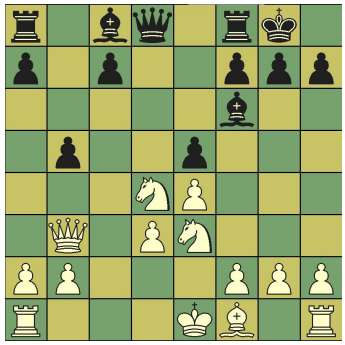
Les mostres que s'executen formen part totes d'una sola partida que vaig jugar contra el *GNU Chess*. Es va guardar en 9 fitxers 9 moments d'aquesta partida, anotant el moviment que vaig portar a terme i la resposta del *GNU Chess*. Es van

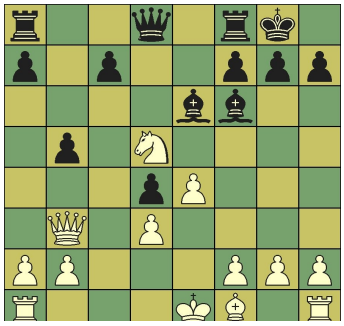
repartir els fitxers de manera que es pot dir que els 2 primers formen part del inici de la partida, els 5 següents formen part del desenvolupament de la partida i els 2 últims als moments finals de la partida. No cal dir que la partida la va guanyar el *GNU Chess*. Aquestes 9 mostres són les següents:

Fitxer de dades	Moviment	Estat de la partida
	**d4**	
	**B1d2**	
	**C4e3**	

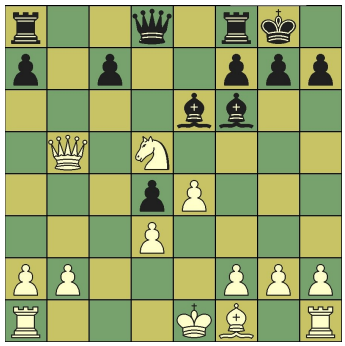


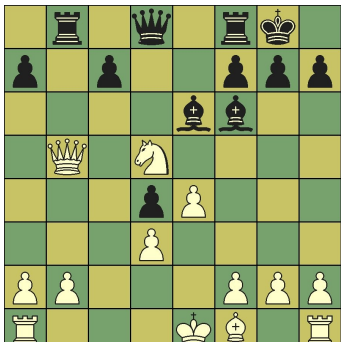
Nxd4



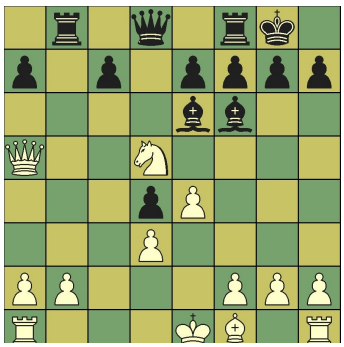


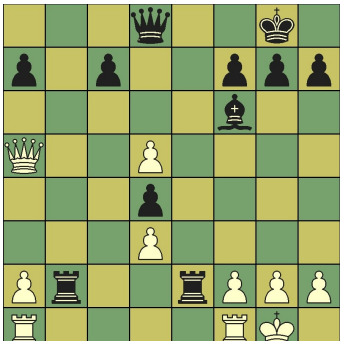
Qxb5



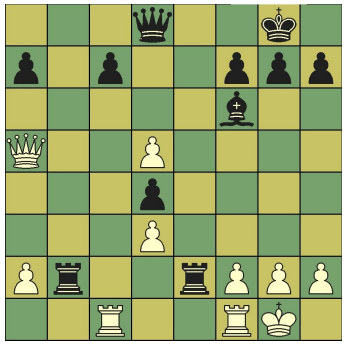


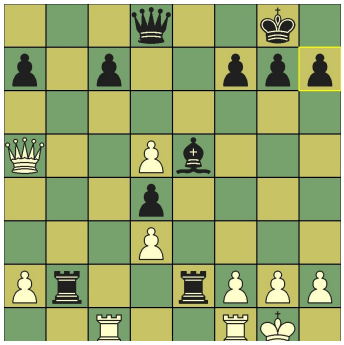
b5a5



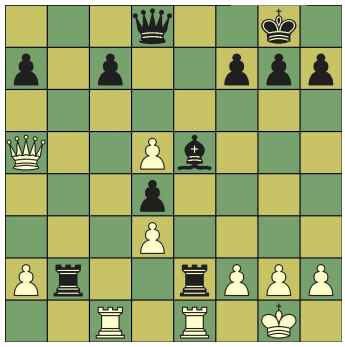


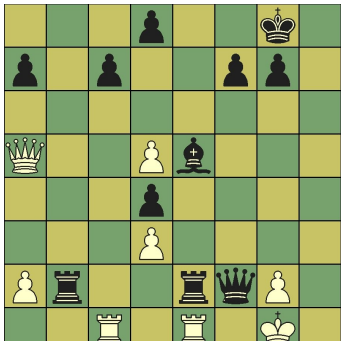
a1c1



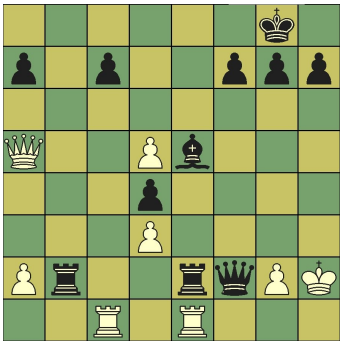


f1e1





g1h2



En la primera columna tenim l'estat de la partida quan carreguem el fitxer, en la segona el moviment que fem i en la tercera el moment en què queda la partida un cop fet el moviment. La resposta de *GNU Chess* en cada cas depèn **exclusivament** de la profunditat de l'arbre que estem executant. Suposem que a profunditat major el moviment és millor, si més no, més estudiat.

La resposta de *GNU Chess* **no depèn** de l'algorisme que l'executa, això significa que crearem mètriques segons el temps de resposta i el nombre de nodes calculats per arribar a ella.

Cada conjunt d'aquests moviments l'anomenarem un **grup de mostres.** Recollirem per a cada als algorismes Iteratiu i Exhaustiu(Paral·lel) un grup de mostres per a arbres de profunditat 5, 6, 7, 8, 9, 10 i 11. I per al algorisme mixt recollirem un grup

de mostres per a les profunditats 8, 9, 10, 11 (ja que un mixt de menys de 8 nivells de profunditat correspon a un iteratiu!). Tindrem, doncs, 18 grups de mostres a avaluar. A 9 moviments per grup fan un total de 162 mostres.

Creiem que ja serà suficient per a fer-nos una idea de les característiques de cada un.

Mostrarem per a cada algorisme una taula amb els resultats i els gràfics corresponents al temps emprat per a cada moviment i el nombre de nodes avaluats en cada cas. La interpretació dels resultats, així com les mitjanes i conclusions numèriques ho deixem per a l'ultim apartat, un cop hàgim posat de relleu tots els valors obtinguts.

S'ha de tenir en compte que la configuració de *Condor,* preparat per a respondre les peticions fins a l'ordre de minuts, i la sobrecarrega de la màquina *aocegrid* suposa un temps 'real' molt gran en l'execució d'aquestes proves. Hem obviat aquest factor i només mostrarem els resultats d'aquests algorismes segons el seu temps d'execució.

5.2. Dades Algorisme Iteratiu.

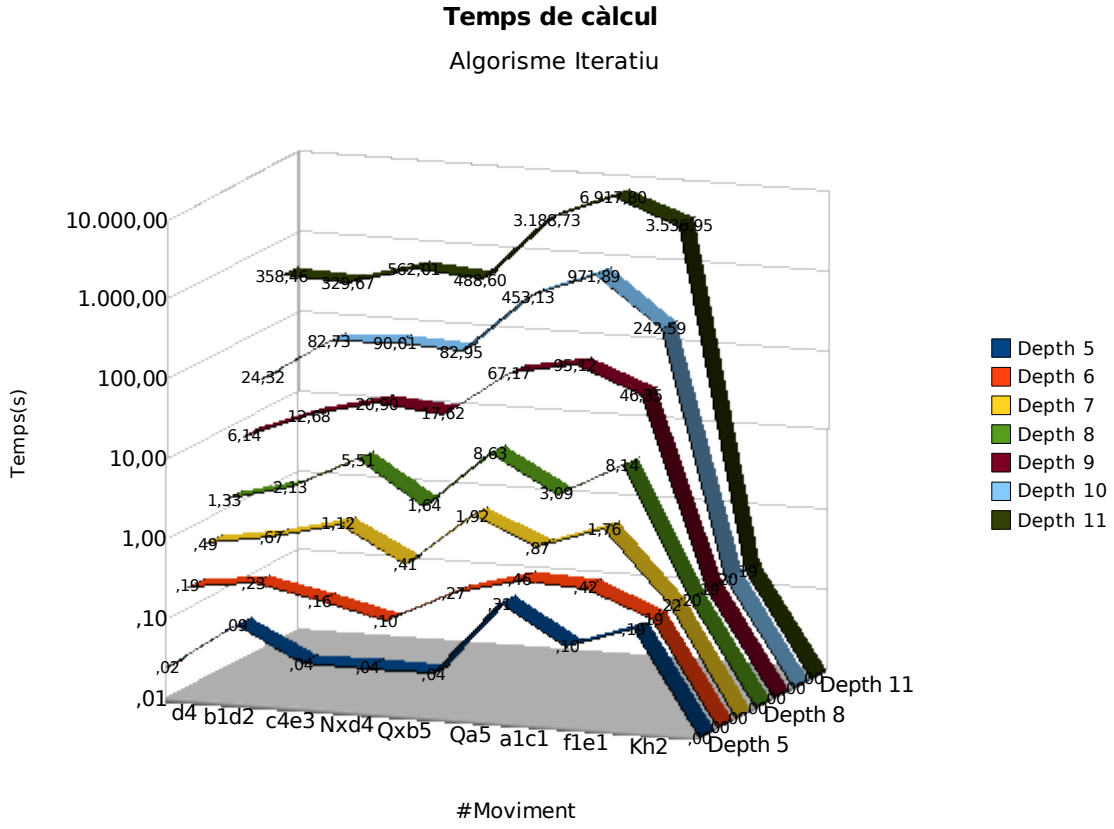
Comencem per avaluar l'algorisme que ens dóna *GNU Chess* per defecte.

	d4	b1d2	c4e3	Nxd4	Qxb5	Qa5	a1c1	f1e1	Kh2
Depth	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul
5	0,023884	0,094151	0,035945	0,035782	0,036129	0,313720	0,102740	0,192289	0,000054
6	0,185786	0,232311	0,162840	0,104119	0,268449	0,459511	0,418926	0,193163	0,000091
7	0,485681	0,670700	1,115858	0,409948	1,915654	0,873729	1,758537	0,219393	0,000058
8	1,328769	2,125524	5,507711	1,640410	8,631765	3,085575	8,144718	0,201151	0,000133
9	6,142582	12,682428	20,902886	17,620123	67,172642	95,121418	46,345378	0,192921	0,000057
10	24,319428	82,733259	90,006281	82,947970	453,125969	971,888928	242,591631	0,200867	0,000138
11	358,464078	329,673827	562,011229	488,596337	3188,729488	6917,798678	3536,948324	0,192084	0,000082

Esquema 1: *temps de càlcul per a cada moviment segons la profunditat*

La distribució del temps de càlcul de l'arbre augmenta exponencialment en cada valor de profunditat. Es per això que hem cregut necessari presentar el gràfic en un rang de valors exponencial per a poder apreciar tots els valors de cada profunditat. D'altra els valors de profunditats més baixes

Aquest és el gràfic:



Cada mostra(Columna Z) del gràfic correspon a una profunditat diferent. A cada columna Y (moviments) li correspon un valor de la columna X (temps dedicat al càlcul)

Tal i com havíem pensat l'augment del temps és exponencial i totes les profunditats segueixen el mateix patró: a mesura que anem avançant en la partida es va augmentant progressivament el temps de càlcul fins a arribar a un màxim. A partir

d'allà el temps de partida descendeix considerablement fins a arribar als pràcticament 0 segons al arribar al moviment final.

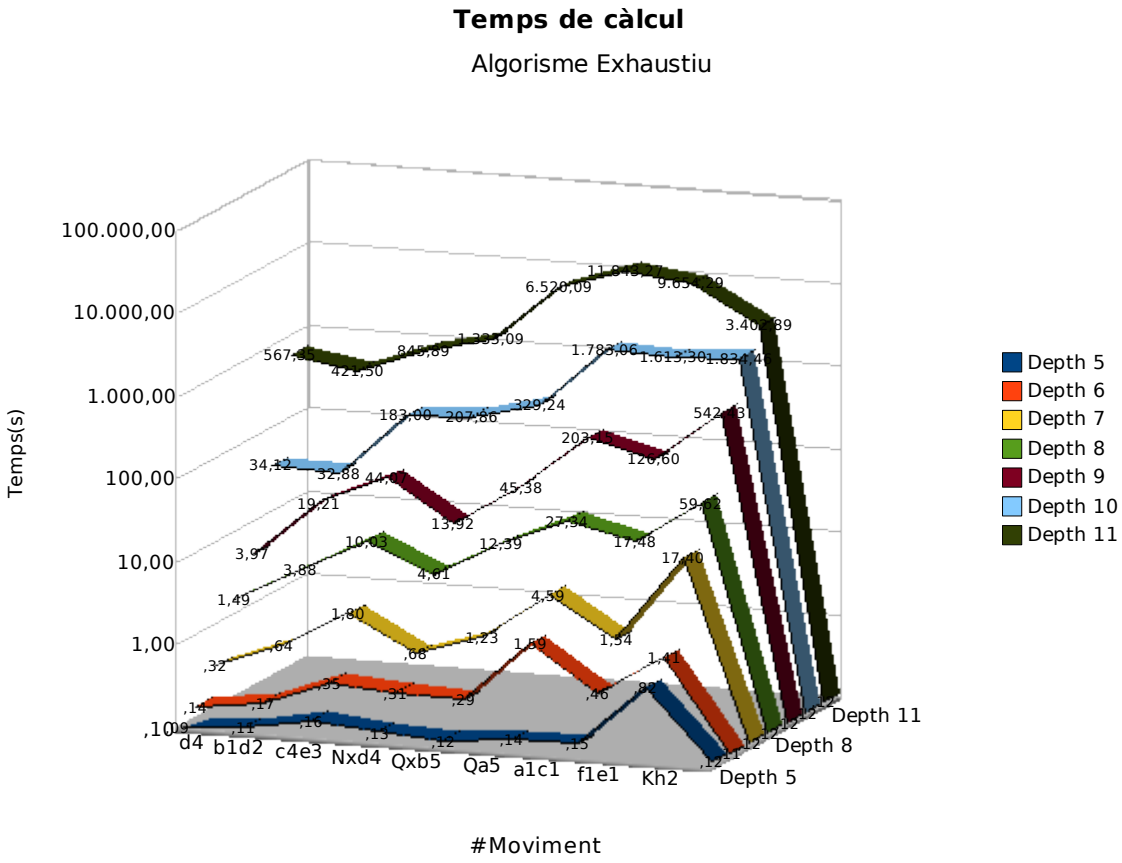
Aquest descens es deu a una característica de *GNU Chess* que no havíem parlat fins ara: al generar els moviments, *GNU Chess* genera primer els moviments que signifiquen una captura de les peces contràries perquè, sobre el paper, són els que ens donaran millors resultats. D'aquests, el primer que té en compte es la captura del rei i en conseqüència, final de la partida. Al avaluar aquests moviments finals, *GNU Chess* considera que no farà falta avaluar res més, que ja en té prou. Com que hem generat aquests moviments en primer lloc, no li costa massa calcular la jugada. (Notar en la taula, que en l'ultim moviment només s'avalua un node!)

5.2. Dades Algorisme Exhaustiu.

Farem el mateix pel l'algorisme exhaustiu:

	d4	b1d2	c4e3	Nxd4	Qxb5	Qa5	a1c1	f1e1	Kh2
Depth	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul
Depth 5	0,091389	0,113736	0,160322	0,134855	0,123755	0,143961	0,149616	0,824104	0,121261
Depth 6	0,135327	0,173206	0,345953	0,307888	0,291472	1,590800	0,459241	1,408106	0,114000
Depth 7	0,315358	0,636210	1,795966	0,679705	1,233665	4,593901	1,540677	17,395303	0,116529
Depth 8	1,488379	3,882354	10,025990	4,606819	12,387861	27,338309	17,481466	59,620193	0,115756
Depth 9	3,972584	19,210198	44,066562	13,919965	45,382161	203,149896	126,604178	542,425898	0,115690
Depth 10	34,123019	32,878433	182,998831	207,857596	329,235526	1783,061237	1613,301124	1834,456023	0,115745
Depth 11	567,348291	421,502837	845,887209	1333,093721	6520,087377	11843,265438	9654,294780	3402,889011	0,117802

i extraurem el mateix tipus de gràfic per mostrar les seves dades:



Veiem que segueix el mateix augment exponencial, però aquesta vegada els valors de temps encara són superiors. Aquesta ha sigut una sorpresa que ens vam endur. Pensàvem que el fet de tenir una execució distribuïda, encara que oferís un algorisme que retallava menys branques, ens asseguraria una millor resposta temporal. Tanmateix, no ha sigut així. El nombre de nodes calculats ha sigut tan gran, que ni tant sols distribuint el treball hem pogut millorar la rapidesa.

Podem veure que aquí, el valor decau en l'ultim moviment enlloc dels dos últims. Això es deu al fet que en l'ultim moviment hi hem aplicat una petita millora que comprova si el primer moviment es jac mat. Si és així, no envia l'execució paral·lela.

Si no haguéssim aplicat aquesta millora, ens trobaríem que l'últim moviment és tant o més complicat que els anteriors, ja que la resta, al executar-se en màquines

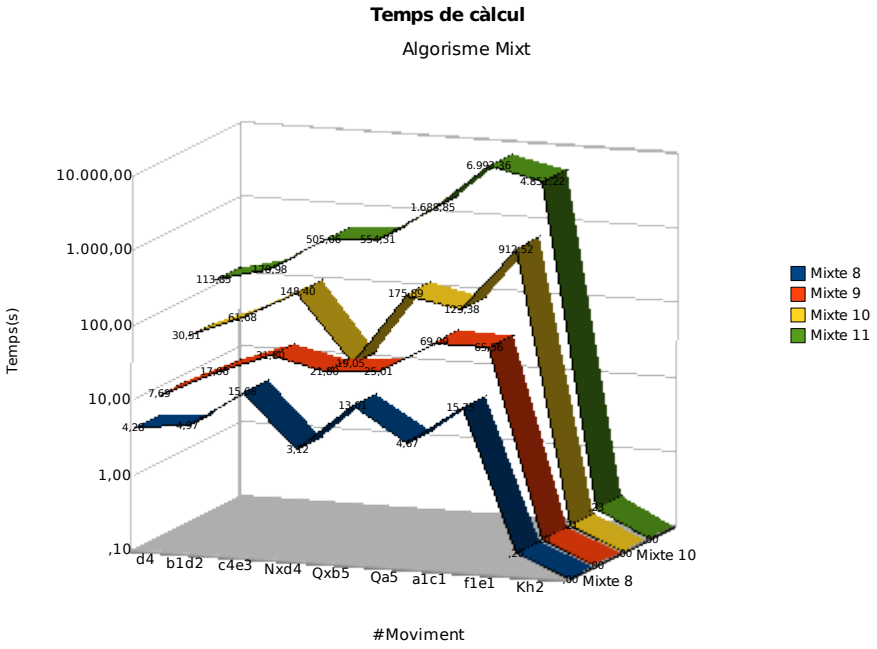
diferents, no coneixen que el primer d'ells resulta ser el final de la partida.

5.3. Dades Algorisme Mixt

I un cop més, el mateix per a l'algorisme mixt. Aquesta vegada, però , el nombre de mostres és inferior, ja que amb menys de 8 nivells de profunditat en realitat tenim l'algorisme iteratiu. Tan de bo s'hagués tingut temps per a poder fer més proves, però el temps del que es va disposar va ésser limitat.

	d4	b1d2	c4e3	Nxd4	Qxb5	Qa5	a1c1	f1e1	Kh2
Depth	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul	Temps de calcul
Mixte 8	4,281613	4,970050	15,678233	3,117895	13,014403	4,670042	15,750071	0,201453	0,000093
Mixte 9	7,688281	17,058271	31,795662	21,795662	25,012700	69,089768	65,561849	0,202951	0,000114
Mixte 10	30,510204	61,677271	148,402179	19,047170	175,893870	123,377800	912,522903	0,211324	0,000066
Mixte 11	113,852666	178,979671	505,055979	554,308958	1688,850000	6993,357277	4851,219300	0,230731	0,000101

El gràfic, seguint el mateix esquema que els dos anteriors, és el que ve a continuació:



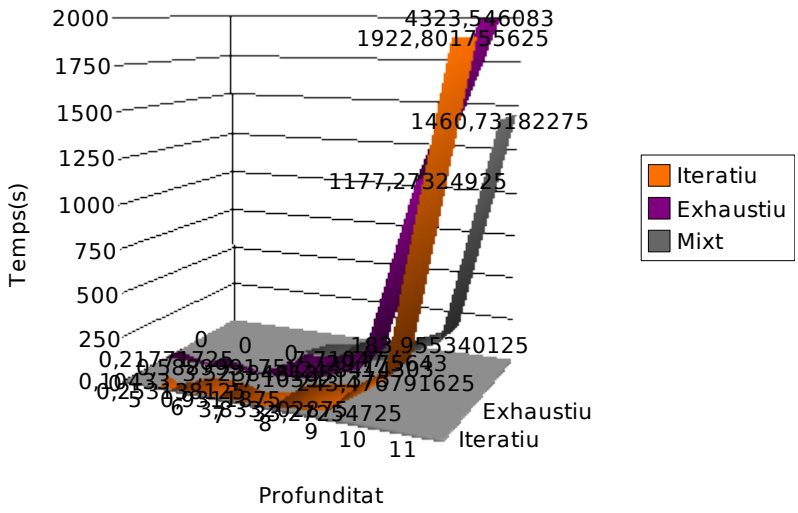
És un gràfic semblant als dos anteriors però arriba a nivells de temps generalment inferiors. Recupera la característica del algorisme iteratiu de començar a baixar en

els dos últims moviments (crec que amb un parell de mostres més entre el **a1c1** i el **f1e1** no veuríem una tendència tan brusca).

5.5. Comparativa d'algorismes.

El nombre de proves utilitzades fa que les conclusions, encara que siguin fiables, no acaben de ser massa acurades. Intentarem treure'n el màxim de joc possible fent una comparativa entre la mitjana de temps de càlcul de cada profunditat i fent una valoració posterior.

Mitjana de temps per algorisme



Cada una de les línies correspon a un algorisme diferent. L'eix de les X correspon a cada una de les profunditats que hem calculat i l'eix de les Y al temps que tarda de mitjana cada un dels algorismes. Hem escollit un format 3D perquè els valors de les profunditats 5, 6, i 7 són molt semblants per als algorismes iteratiu i exhaustiu i ho hem preferit així perquè no es sobreescriguin els valors. Es pot veure una tendència

exponencial molt marcada a partir de profunditat 9. Tant marcada que hem retallat l'algorisme a 2000 segons com a màxim deixant fora l'ultim valor del algorisme exhaustiu, que arribava a més dels 4000 segons.

Com podem veure, l'algorisme que ens dóna millor resposta temporal és el mixt. Hem d'afegir a això que la part *local* és el coll d'ampolla de l'algorisme, i que en una màquina menys carregada que la que vam utilitzar i amb alguns ordinadors més executant treballs a través de *Condor* milloraria moltíssim les seves prestacions.

5.6. Conclusions

El que en podem extreure, doncs d'aquests gràfics, és que el fet de donar molta potència de càlcul a través d'un sistema distribuït no implica una millora de la rapidesa de càlcul, si darrera no té un bon algorisme. Ja hem demostrat en l'apartat 2.2.3.4 que la combinació *PVS+Iterative Deepening* és molt més ràpida i eficient que un *Min-Max* amb poda *alpha-beta*. En aquest cas, inclús executant cada un dels fills en màquines diferents, seguim tardant molt més en calcular-ho. Haig de reconèixer que aquest resultat m'ha sorprès.

Ha sigut l'execució Mixta, la que prèviament ordena l'arbre i aprofita l'algorisme *PVS*, la que ens dóna més bon resultat a mesura que es va augmentant la profunditat de cerca. Els resultats parlen per si sols.

El fet de combinar una part en *local* amb una part en paral·lel en l'algorisme mixt també ens evita fer un estudi sobre en quin moment de la partida seria millor deixar de aplicar la computació paral·lela, ja que un bon moviment al final de la partida ja talla l'avaluació de nodes abans d'arribar a una profunditat major a 7 i fa aquesta feina per nosaltres, com hem vist en la última jugada.

A més del temps de càlcul, hem de trobat una avantatge extra del algorisme Mixt

sobre els altres dos: l'escalabilitat. Per suposat, el primer cas només es pot executar en una màquina. L'algorisme exhaustiu limita el seu nombre de execucions al nombre de fills del node arrel. Així, encara que tinguem 500 màquines disponibles, només n'aprofitaríem unes 35 com a màxim. En canvi, l'algorisme mixt estén la seva escalabilitat al nodes de segon nivell, de manera que es pot executar en unes 1225 (35*35) màquines aprofitant-les totes. Dubtem de que algú mai disposi de tantes màquines per jugar a escacs.

I no només això. Si s'observen els arxius de *logs*, es pot veure que el coll d'ampolla d'aquest algorisme és la part que s'executa en local, ja que el 80% del temps dedicat és calcula en la pròpia màquina. Això ho tenim enviant uns 70 fills a *Condor.* Si el nombre de fills augmenta, els temps de càlcul en respecte dels altres dos algorismes es veurà significativament reduït.

I crec doncs, que tenint en en compte que aquest és l'objectiu *del grid computing,* t, podem dir que els objectius marcats en l'inici d'aquesta memòria s'han complert satisfactòriament.

Epíleg

Arribem aquí a la fi del projecte. És en aquest punt on un fa balanç sobre la feina feta i es pregunta si els objectius enfocats al principi s'han complert.

L'objectiu inicial d'aquest projecte era crear una versió del *GNU Chess* que es pogués executar a través del sistema *Condor*. Si ens ho mirem estrictament s'ha aconseguit. Però el fet d'estudiar l'algorisme del *GNU* Chess i veure com responia la seva adaptació *a Condor* ens va portar a elaborar un segon algorisme per a millorar el primer... i això ja va canviar una mica la perspectiva. No es tractava tant d'aconseguir que s'executés, sinó que millorés la seva resposta temporal, cosa que en principi es donava per fet.

Crec que una bona estimació del temps i un treball no molt intens però constant van permetre arribar amb temps a reparar els continus errors, estudiar millor l'algorisme inicial (sobretot la part d'ordenació) i a poder fer aquesta segona evolució.. i a estar més convençuts que la feina s'havia fet *realment.*

El tema de les proves ha sigut un punt i apart. Degut a circumstàncies que ja hem apuntat, el banc de proves ha quedat limitat. A més l'ultima setmana han sorgit alguns errors de codi inesperats, hi han hagut problemes amb la màquina que executava els treballs... una mica de tot. M'hagués agradat poder-les fer amb més mostres, repetir-les... i això hagués portat a un anàlisi més acurat. Hagués donat temps a fer un apartat de possibles millores a fer (com a mínim a enumerar-les) per a poder augmentar les prestacions del algorisme mixt.

Tot i això, el *GNU Chess* ha estat estudiat, la evolució marcada ala principi s'ha fet,

hem 'matxacat' al sistema de *Condor i* a la pobre màquina d'*aocegrid*, i s'ha millorat en resposta temporal l'algorisme original. Crec que els objectius s'han complert.

Anexos

Annex A. Fitxers de configuració local de Condor.

A l'entorn local.

```
COLLECTOR_NAME =
 FILESYSTEM_DOMAIN = $(FULL_HOSTNAME)
 SUSPEND =
 LOCK = /tmp/condor-lock.$(HOSTNAME)0.531608855658309
 JAVA_MAXHEAP_ARGUMENT =
 CONDOR_ADMIN = jaumedevesa@gmail.com
 MAIL = /usr/bin/mail
 START = TRUE
 RELEASE_DIR = /home/jdevesa/Condor/release_dir
 DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, SCHEDD, STARTD
 COLLECTOR = $(SBIN)/condor_collector
 PREEMPT = FALSE
 UID_DOMAIN = $(FULL_HOSTNAME)
 EMAIL_DOMAIN = $(FULL_HOSTNAME)
 NEGOTIATOR = $(SBIN)/condor_negotiator
 JAVA = /usr/bin/java
 VACATE = FALSE
 SUSPEND = FALSE
CONDOR_HOST = $(FULL_HOSTNAME)
CONDOR_IDS = 1000.100
 LOCAL_DIR = /home/jdevesa/Condor/local_dir
 HOSTALLOW_READ = *
 HOSTALLOW_WRITE = *
 LOWPORT = 40000
 HIGHPORT = 50000
```

```
RESERVED_SWAP = 0
```

Annex B. Excel amb les proves finals

	Moviment 1 --> d4				
	Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme
Iteratiu 5	0,023884	8994	9193	Nf6	-7
Exhaustiu 5	0,091389	50450	52485	Nf6	-7
Mixte 5	0,000000	0	0	---	---
Iteratiu 6	0,185786	74723	79331	e6	-49
Exhaustiu 6	0,135327	189463	206766	Nf6**	-40
Mixte 6	0,000000	0	0	---	---
Iteratiu 7	0,485681	194552	208784	d5*	-10
Exhaustiu 7	0,315358	1127637	1245604	d5*	-10
Mixte 7	0,000000	0	0	---	---
Iteratiu 8	1,328769	531438	627404	d5*	-34
Exhaustiu 8	1,488379	3904048	4726797	Nf6**	-34
Mixte 8	4,281613	11764587	12937741	d5*	-34
Iteratiu 9	6,142582	2481575	2955045	d5*	-20
Exhaustiu 9	3,972584	18932084	23819045	d5*	-20
Mixte 9	7,688281	13851282	17301885	d5*	-20
Iteratiu 10	24,319428	9812251	11627337	d5*	-36
Exhaustiu 10	34,123019	146809200	187429180	d5*	-34
Mixte 10	30,510204	58266436	78175959	d5*	-36
Iteratiu 11	358,464078	64471021	80724880	e6	-22
Exhaustiu 11	567,348291	1827390374	2462718239	e6	-22
Mixte 11	113,852666	409434125	496920842	e6	-22

Moviment 2 --> Nb2					Moviment 3 --> Ne3				
Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme	Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme
0,094151	27210	46185	d5	34	0,035945	13772	17527	O-O	97
0,113736	170702	232954	d5	34	0,160322	812829	1112826	O-O	97
0,000000	0	0	---	---	0,000000	0	0	---	---
0,232311	87371	145920	d5	38	0,162840	70945	89160	O-O	56
0,173206	609413	814463	d5	38	0,345953	3526801	4459925	O-O	56
0,000000	0	0	---	---	0,000000	0	0	---	---
0,670700	252700	436447	d5*	43	1,115858	459231	599682	O-O	61
0,636210	1785587	2752328	d5*	43	1,795966	17470328	27317054	O-O	61
0,000000	0	0	---	---	0,000000	0	0	---	---
2,125524	913552	1444895	d5*	51	5,507711	2332427	3252185	Be6	52
3,882354	7552326	11193758	d5	51	10,025990	89187384	126416783	Be6	51
4,970050	15116031	25304010	d5*	51	15,678233	42089704	56473910	Be6	52
12,682428	5702051	8512605	d5*	22	20,902886	8952687	11931934	Be6	58
19,210198	45200061	64563511	d5*	22	44,066562	651024698	1016246026	Be6	57
17,058271	40755704	65186638	d5*	22	31,795662	43490814	57415266	Be6	58
82,733259	9812251	49343554	d5*	20	90,006281	38912802	52713083	Be6	47
32,878433	141294944	183809781	d5*	20	182,998831	3295234951	5052087401	Be6	47
61,677271	169233760	265934985	d5*	20	148,402179	257535161	354660017	Be6	47
329,673827	163982326	203346161	d5	25	562,011229	264382597	373864885	Be6	51
421,502837	1530285445	2166092231	d5	25	845,887209	5882298476	8812093889	Be6	51
178,979671	569928477	870458322	d5	25	505,055979	970540333	1355648065	O-O**	51

Moviment 4 --> Nxd4					Moviment 5 --> Qxb5				
Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme	Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme
0,035782	17815	28249	exd4	47	0,036129	35498	54272	Rb8	84
0,134855	793518	1230751	exd4	45	0,123755	374880	494414	Rb8	84
0,000000	0	0	---	---	0,000000	0	0	---	---
0,104119	50299	83124	exd4	39	0,268449	132567	209162	Rb8	78
0,307888	2554318	3594618	exd4	39	0,291472	1288045	1540649	Rb8	78
0,000000	0	0	---	---	0,000000	0	0	---	---
0,409948	198086	362414	exd4	35	1,915654	929640	1472383	Rb8	64
0,679705	13728503	23867801	exd4	35	1,233665	5234484	6980087	Rb8	62
0,000000	0	0	---	---	0,000000	0	0	---	---
1,640410	783007	1353660	exd4	27	8,631765	4242965	6908457	Rb8	58
4,606819	107534448	179710971	exd4	27	12,387861	24232873	33579011	Rb8	58
3,117895	12400212	846383	exd4	27	13,014403	110870856	179882428	Rb8	58
17,620123	8770119	14193368	Qxd4	20	67,172642	33036834	54243907	Rb8	72
13,919965	131433285	205636413	Qxd4	20	45,382161	137909184	197055611	Rb8	72
21,795662	26833608	42055603	Qxd4	20	25,012700	693857097	1157317085	Rb8	72
82,947970	38912802	63821766	exd4	30	453,125969	220711506	375002255	Rb8	69
207,857596	2620438223	4182336937	exd4	29	329,235526	753677443	1099120007	Rb8	69
19,047170	85354386	130507129	exd4	30	175,893870	4085005316	6905857164	Rb8	69
488,596337	254901542	400793808	Qxd4	14	###	1583659473	2627844791	Rb8	64
###	###	###	Qxd4	14	###	###	###	Rb8	63
554,308958	393400262	585830819	Qxd4	14	###	###	###	Rb8	64

Moviment 6 --> Qa5					Moviment 7 --> Rac1				
Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme	Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme
0,313720	147943	227915	bxd5	152	0,102740	50117	90318	Be5	611
0,143961	2142602	3551836	bxd5	150	0,149616	1156409	1938796	Be5	611
0,000000	0	0	---	---	0,000000	0	0	---	---
0,459511	227600	319139	bxd5	211	0,418926	223997	386972	Be5	592
1,590800	13489565	21826721	bxd5	211	0,459241	5016422	7114147	Be5	592
0,000000	0	0	---	---	0,000000	0	0	---	---
0,873729	458501	662920	bxd5	220	1,758537	916700	1602975	Be5	588
4,593901	55129344	94102409	bxd5	220	1,540677	21573761	33995436	Be5	588
0,000000	0	0	---	---	0,000000	0	0	---	---
3,085575	1619291	2413823	bxd5	232	8,144718	4171296	7228622	Be5	603
27,338309	401800383	681711738	bxd5	232	17,481466	50348782	82547613	Be5	603
4,670042	20242982	27786574	bxd5	232	15,750071	137560477	246655709	Be5	603
95,121418	51118673	85498273	bxd5	229	46,345378	25276003	42033594	Be5	664
203,149896	2419293233	4246125213	bxd5	229	126,604178	593688630	925872944	Be5	664
69,089768	120164522	198989628	bxd5	229	65,561849	517773389	893307985	Be5	664
971,888928	522878121	881649393	bxd5	253	242,591631	522878121	220410306	Be5	649
###	###	###	bxd5	251	###	6324942794	6941111823	Be5	649
123,377800	985698750	1905145397	bxd5	253	912,522903	4274841081	6769753848	Be5	649
###	3736455577	1890242346	bxd5	248	###	2225516741	3455668798	Be5	635
###	###	###	bxd5	248	###	###	###	Be5	635
###	###	###	bxd5	248	###	###	###	Be5	635

Moviment 8 --> Rfe1					Moviment 9--> Kh2				
Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme	Temps de calcul	# de nodes evaluats	# de nodes generats	Resposta del sistema	Resultat Algorisme
0,192289	114381	207572	bxh2	32759	0,000054	1	74	Qxg2#	32767
0,824104	2536300	5118048	bxh2	32759	0,121261	1	74	Qxg2#	32767
0,000000	0	0	---	---	0,000000	0	0	---	---
0,193163	114381	207572	bxh2	32759	0,000091	1	74	Qxg2#	32767
1,408106	12474621	26690976	bxh2	32759	0,114000	1	74	Qxg2#	32767
0,000000	0	0	---	---	0,000000	0	0	---	---
0,219393	114381	207572	bxh2	32759	0,000058	1	74	Qxg2#	32767
17,395303	512027633	1111139998	bxh2	32759	0,116529	1	74	Qxg2#	32767
0,000000	0	0	---	---	0,000000	0	0	---	---
0,201151	114381	207572	bxh2	32759	0,000133	1	74	Qxg2#	32767
59,620193	7231054903	15546472513	bxh2	32759	0,115756	1	74	Qxg2#	32767
0,201453	114381	207572	bxh2	32759	0,000093	1	74	Qxg2#	32767
0,192921	114381	207572	bxh2	32759	0,000057	1	74	Qxg2#	32767
542,425898	?	?	bxh2	32759	0,115690	1	74	Qxg2#	32767
0,202951	114381	207572	bxh2	32759	0,000114	1	74	Qxg2#	32767
0,200867	114381	207572	bxh2	32759	0,000138	1	74	Qxg2#	32767
1234,730228	785607543	1329480802	bxh2	32759	0,115745	1	74	Qxg2#	32767
0,211324	114381	207572	bxh2	32759	0,000066	1	74	Qxg2#	32767
0,192084	114381	207572	bxh2	32759	0,000082	1	74	Qxg2#	32767
3402,889011	922836640	15888920978	bxh2	32759	0,117802	1	74	Qxg2#	32767
0,230731	114381	207572	bxh2	32759	0,000101	1	74	Qxg2#	32767

Totals				
Mitjana de temps	Mitjana de nodes evaluats	Mitjana de nodes generats	Relació generats / evaluats	Nodes calculats per segon
0,104330	51966,25	85153,88	61,03%	498094,99
0,217717	1004711,25	1716513,75	58,53%	4614752,62
0	0	0	0,00%	0,00%
0,253138	122735,38	190047,5	64,58%	484855,35
0,588999	4893581	8281033,13	59,09%	8308298,56
0	0	0	0,00%	0,00%
0,931188	440473,88	694147,13	63,46%	473023,83
3,523848	78509659,63	162675089,63	48,26%	22279524,21
0	0	0	0,00%	0,00%
3,833203	1838544,63	2929577,25	62,76%	479636,66
17,103921	989451893,38	2083294898	47,49%	57849417,78
7,710470	43769903,75	68761790,88	63,65%	5676684,27
33,272547	16931540,38	27447037,25	61,69%	508874,19
124,841430	499685146,88	834914845,38	59,85%	4002558,65
29,775643	182105099,63	303972707,75	59,91%	6115908,21
243,476792	170504029,38	206846908,25	82,43%	700288,63
1177,273249	4280471717,13	5785685394,13	73,98%	3635920,31
183,955340	1239506158,88	2051280258,88	60,43%	6738081,96
1922,801756	1036685457,25	1129086655,13	91,82%	539153,58
4323,546083	24971719123,38	37097486990,13	67,31%	5775749,5
1860,731823	11504583680,13	13626088771	84,43%	6182827,39

RESUM

ULTRA DEEP BLUE: THE ULTIMATE CHESS PLAYER és un projecte d'adaptació del programa *GNU Chess* al sistema de *grid computing* 'Condor'. I amb això, es planteja un estudi sobre els algorismes de cerca i la seva aplicació en entorns distribuïts. Una sèrie de proves sobre unes mostres de una partida d'escacs contra el propi *GNU Chess* ens ajuden a posar de relleu els avantatges i inconvenients de cada un dels algorismes proposats.

RESUMEN

ULTRA DEEP BLUE: THE ULTIMATE CHESS PLAYER es un proyecto de adapación del programa *GNU Chess* al sistema de *grid computing* '*Condor*'. I con eso, se plantea un estudio sobre los algoritmos de búsqueda y su aplicación en enornos distribuidos. Una serie de pruebas sobre unas muestras de una partida de ajedrez contra el propio *GNU Chess* nos ayudan a poner de relieve las ventajas e inconvenientes de cada uno de los algoritmos propuestos.

ABSTRACT

ULTRA DEEP BLUE: THE ULTIMATE CHESS PLAYER is a project which adapt GNU Chess software at '*Condor*' grid computing system. Once we have this, a study about search algorithms and its possibilities in distributed environments is raised. A series of samples over a chess game against *GNU Chess* help us to enphasize the advantages an disadvantages of each one of the proposed algorithms.

Jaume Devesa, Setembre 2008
Universitat Autònoma de Barcelona