Team Othello:

Adam Friedlander - ajf5570
Joseph Pecoraro - jjp1820
Nicholas Ver Hoeve - nav5463
Team Website: http://www.cs.rit.edu/~jjp1820/parallel/

Description of Our Problem:

Our goal was to implement a parallel game tree search algorithm, such as MTD(f), for Othello. Game tree searching involves building a tree of all the possible moves at a given game position and evaluating the score for each node. It can be customized to lookahead any number of moves, which correlates to the depth of the game tree.

MTD(f) is a more advanced incarnation of Alpha-Beta (αβ) that is both practical and has the theoretical advantages of SSS*. MTD(f) makes a series of null-window Alpha-Beta calls to converge on the resulting score of a position. It is heavily dependent on the transposition table (a hash table of positions already explored and their known windows), but can handily outperform Alpha-Beta. We will explore sequential and Parallel implementations of both algorithms.

Analysis of The First Research Paper:

> Borovska, P. and Lazarova, M. 2007. Efficiency of parallel minimax algorithm for game tree search. In Proceedings of the 2007 international Conference on Computer Systems and Technologies (Bulgaria, June 14 - 15, 2007). B. Rachev, A. Smrikarov, and D. Dimov, Eds. CompSysTech '07, vol. 285. ACM, New York, NY, 1-6.

This paper covers parallel minimax Alpha-Beta, a close relative of MTD(f). It contains a brief description of game trees and the minimax algorithm, but the focus is on parallelization of that algorithm across standard and hybrid clusters. The results of that parallelization are discussed briefly.

As with many other parallel applications, the manager-worker paradigm is used here. The manager distributes potential moves to the workers for analysis and then performs a reduction to determine the best cost function representing the ideal move. This move is in turn broadcast to the workers and is made in the game. The workers evaluate the series of moves and return the cost function for possible moves of the other player at each step, returning the results to the master. They then receive the broadcast of the move selected by the master.

The application covered in this paper was 4x4 tic-tac-toe. While this is an easy game to understand, we felt that the search space did not have sufficient branching, particularly

after the first few moves have been made. (We would have preferred to see something like chess or checkers used as a benchmark). Nonetheless, this result showed that standard minimax and the Alpha-Beta variant can be successfully parallelized.

A notable result from this paper was the significant gain in speedup for a given number of processes by running those processes on hybrid cluster nodes instead of one process per cluster node. By facilitating faster communication, overhead is reduced, and more time can be spent on useful computation. This can be contrasted with problems such as encrypted key space search, where there is little need for communication across nodes and therefore little difference between an *n*-way multiprocessor machine and a cluster of *n* machines with similar configurations.

Analysis of The Second Research Paper:

> Plaat, A., Shaeffer, J., Pijls, W., and de Bruin, A. 1996. Best-first fixed-depth minimax algorithms. Artif. Intell. 87, 1-2 (Nov. 1996), 255-293. DOI= http://dx.doi.org/10.1016/0004-3702(95)00126-3

This paper is the original paper presenting the MTD(f) algorithm. The authors give an in depth examination of the theory behind existing minimax searching algorithms such as Alpha-Beta and SSS* and show how MTD(f) relates to them outperforms them. The paper is heavy on both theory and history but it does an excellent job at explaining MTD(f).

The authors start by presenting SSS*, an algorithm that was proven to evaluate fewer leaf nodes then the industry standard Alpha-Beta. Although it was theoretically superior, it wasn't used in practice for of a number of reasons. It was complex to understand, has large memory requirements, and requires the overhead of maintaining a single large list. The authors claim that they were able to solve these problems, and that in fact the complex SSS* algorithm is merely a special case of an Alpha-Beta algorithm with a Transposition Table. This is the key point argued in the paper and the reason we choose to experiment with the superior algorithm, MTD(f) in our project.

Memory-enhanced null-window Alpha-Beta search is a new concept introduced by the authors. The idea is that there is a driver that makes a number of repeated calls to this procedure in order to converge on the final minimax value. A transposition table built up in the previous calls can be reused in the successive calls. This concept is nicknamed MT for Memory-enhanced Test. The MT concept is fundamental to MTD(f).

The authors reformulate a number of the existing algorithms, including SSS*, to use the MT concept. They give a detailed examination of how they would transform an SSS* algorithm into their MT-SSS* by using an underlying Alpha-Beta algorithm. They performed tests to see if SSS*'s large memory requirements were eliminated. There are numerous graphs showing the results of their tests. Essentially, their version did indeed fit in memory, and given enough memory for their transposition table, they

outperformed the industry standard Alpha-Beta algorithm. Also, because they reformulated their algorithm in terms of Alpha-Beta calls, the cost of transforming an existing Alpha-Beta algorithm to their improved algorithm was relatively low.

The idea that almost all existing Alpha-Beta algorithms could simply be wrapped inside a “driver” such as MT-SSS* and achieve a performance increase is a powerful one. This means that existing algorithms could benefit from the theoretical improvements that the complex SSS* algorithm offered, but keep their practical and fully functional Alpha-Beta implementation. In our implementation we focused on building an Alpha-Beta algorithm for Othello. MTD(f) and our parallel version both sat on top of this Alpha-Beta implementation.

In the next section of the paper the authors introduce their framework for MTD(f) algorithms. They present a number of different algorithms, experimenting with different bounds which they point out may be optimal for different problem domains. They provide the framework for the basic MTD(f) algorithm that makes use of an underlying algorithm and describe its performance results.

The performance results for MTD(f) are compared against many of the previously mentioned algorithms but also some of the industry standard algorithms such as Aspiration NegaScout. MTD(f) consistently evaluated less leaf nodes, and less total nodes then all of the other algorithms. Its execution time improved between 5% and 16% on some existing test programs.

The paper gives us the pseudocode we need to increase the performance of an existing Alpha-Beta implementation. It gives the theory and the practical results that back up their claims for its improvements. Not only did the paper help us implement an efficient algorithm, but it helped us identify what was important to focus on, such as having a strong underlying Alpha-Beta algorithm with an efficient transposition table.

Analysis of The Third Research Paper:

> Kishimoto, A. and Schaeffer, J. 2002. Distributed Game-Tree Search Using Transposition Table Driven Work Scheduling. In Proceedings of the 2002 international Conference on Parallel Processing (Icpp'02) (August 18 - 21, 2002). ICPP. IEEE Computer Society, Washington, DC, 323.

This paper focused on the author’s attempt to improve the performance of parallel algorithms in a two player game by using a successful concept in the single player domain. That concept was Table Driven Scheduling (TDS). In the paper the authors identified problems that they identified while making their parallel algorithms and their ideas and final solutions to the problems. They included an outline of their final algorithm and its resulting benchmarks when implemented for a few modern games.

The authors start out by explaining what they hoped to accomplish with their research. They mention that, shortly before the time their paper was written, there were two important developments to game searches. First, MTD(f) algorithm was emerging as the new standard. Second, TDS had proven to be a powerful paradigm for distributed single-agent game searches. They claimed that their work was the first attempt to combine both concepts at the same time in a two player game. We would have to agree that MTD(f) is indeed now the standard game search algorithm.

The “Table” part of TDS is essentially a memoization table for the results of previously calculated game positions. This is used to prevent repeated processing, which would be wasted time. We had seen this concept show up in one of the other papers and on numerous online resources.

MTD(f) is described by the authors as “the most efficient variant of sequential αβ.” They briefly explain show the algorithm and explain that at its core it is just making subsequent calls to an underlying αβ algorithm. They point out that due to its repeated processing of the entire tree with different search windows it relies heavily on the transposition table to prevent repeated work. Throughout the paper they refer to their algorithms as working with any αβ algorithms, therefore including MTD(f). Up to this point the descriptions given by the authors were exactly in line with how we felt about parallel game tree searching. The algorithm they showed were exactly what we expected.

At this point the authors point out a number of the “obstacles to parallel search” that must be dealt with in order to produce a working and efficient parallel program. The obstacles that they mention are with respect to a Cluster parallel implementation however much of it is applicable in our SMP implementation. The authors mentioned general parallel concerns such as search overhead, synchronization, communication overhead, and load balancing. Although some of their descriptions were general, the authors also went one step further and specified how each obstacle specifically related to game tree searching.

Search overhead is the concept that the parallel algorithm will likely process a different number of nodes in the game tree, most likely more but it could potentially be less. In parallel implementations there is a clear potential for repeated work or having a processor perform work that later turned out to be completely unnecessary due to game tree pruning. Our own implementation ran into this problem quite a bit, and we experimented with a few of the solutions proposed by the authors.

Synchronization and load balancing are both related. They both deal with the goal of keeping as few processors idle as possible and thus keeping the work “evenly distributed.” In typical game searching algorithms keeping all the processors busy is a challenge. The authors state that in practice a processor may have to “wait for others to complete their tasks.” This turned out to be a major problem for us with our implementation so their later solutions were useful to us.

Communication overhead deals with the time to send and receive messages. The authors give the example of the importance of fast communication by giving the example that “a late message can result in unnecessary work being spawned in parallel.” This ties into all of the above parallel obstacles. They are all closely related. There will be trade-offs between efficiency and performance. We implemented an SMP version were there was no communication overhead.

The authors mention a few specific parallel search algorithms. They focused on the Young Brothers Wait Concept (YBWC) and explained two of its key features. The first is that the algorithm always tries to order moves from best to worst. That is the order in which it evaluates them. The advantage here is that often the expected best produces a very good cut-off value and thus “reduces the search overhead” by eliminating the need to process many of the other nodes. Secondly, the load balance technique used by YBWC is called work stealing. If a processor is “starved for work” it randomly chooses another processor and steals work for that processor’s work queue. This is eventually the technique that we used in our SMP version. We opted to go with the simplicity of this approach because we had no communication costs that would have injured us had we implemented this on a Cluster.

Next the authors discuss the pros and cons of three different ways to implement transposition tables in a distributed memory environment. The three implementations are named local, partition, and replicated.

Local transposition tables means each processor keeps its own isolated transposition table. There is no communication between the processors for table entires. The advantage with this approach is that there are no communication costs because all table lookups can be done locally. However, the cost is that processors may end up repeating work that may have already been completed by another processor.

Partitioned transposition tables is an attempt to relieve that replication of work by leveraging massages. Each processor will keep a “disjoint subset of the table entries.” When a processor needs to perform a lookup it can send a direct message to the processor that should hold that table entry. Likewise when updating a table entry a message must be sent to the proper processor that stores that entry. A key advantage is that the table size is more scalable and there is less repeated work. There are some communication costs but messages are direct and thus between only two processors.

Replicated transposition tables tries to keep a clone of the same table in each processor. Here, lookups would be still be local because each processor would have the result in its own table but there is a much larger communication cost. Updating table entries means broadcasting the result to all processors. This method has scalability problems both with the size of the table and the number of messages as the number of processors grows.

The benefits of Partitioned tables in a distributed memory environment are clear. It is more scalable than the other approaches both in terms of memory usage and in

minimizing communication overhead. In our SMP implementation we took a shared table approach, but experimented and determined that we could use the authors local table approach when our search algorithm reached a certain depth. We took advantage of this to avoid synchronization in our application, much like the authors wished to avoid communication overhead.

The authors mention that all three approaches may have problems if they encounter Directed Acyclic Graphs (DAG). Situations like this require a certain game position to repeat itself as a result of a series of moves from that game position. Later on the authors avoid this problem by making sure that they store identical positions on the same processor. Identifying this problem early on is important for game search algorithms so that it can be handled in the design. Othello requires the number of pieces on the board to increase by one in every turn, so we avoided this special case. Has we implemented another game, we would have needed to account for it as the authors described.

The authors give an overview of Table Driven Scheduling (TDS). They mention that it makes use of the partitioned table concept. When a tree node needs child nodes evaluated, it distributes that work to the processors that store that table entry. The processor that stores a table entry for a node is considered the “home processor” for that node. In this way the table is driving the distribution of work, thus the “work is moving to the data.” When a home processor needs to evaluate a node, it will most likely already have the associated lookup entries in its own table.

The authors then go into how they would upgrade the existing concepts of Table Driven Scheduling into a two player environment. They briefly go over a number of differences between single player and two player games such as pruning, cut-offs, the search windows, node priority optimizations, and saved history. These differences are minor and straightforward, so they move on to their implementation.

Transposition-table Driven Scheduling Alpha-Beta (TDSAB) is what the authors named their algorithm. They use the TDS work distribution but with three important additions concerning search order, node signatures, and synchronization. The authors want to maintain a best-to-worst sort priority for effective pruning. Each node in the tree has a unique signature that not only identifies that node but also every parent of that node. This is used for effective cut-off messages. The cut-off messages instruct processors to eliminate entire sub-trees of nodes from its queue that are no longer necessary to process. The signature of the root of the now pruned sub-tree is broadcast to all processors. Then, each processor can filter its queue and remove any nodes that have a signature prefix is the same as the signature of the node being cut. Finally, there should be special handling to prevent deadlocks in case of a DAG. We felt that their extension of the original TDS algorithm was a natural, straightforward progression. Their concept of cut-off messages with signatures was very powerful and we made sure to use that in our implementation.

The authors then describe and give pseudocode for their algorithm. It contains three different types of messages: New-Work, Cut-Off, and Search-Result. New-Work messages are incoming work to be added to the processor's queue. Inside the message is a node to process and the id of the home processor that the result should be sent to. Cut-Off messages contain the signature of the root node of a sub-tree that should be pruned. Search-Result messages is the result of a completed task getting sent back to the home processor. We saw the advantages of this design, in that it scalable. Our implementation, although SMP, made heavy use of these concepts.

Each processor is in a loop continually managing incoming messages and evaluating its own work queue. When receiving any of the above messages it modifies its work queue as expected. New-Work messages increase the queue, Cut-Off messages prune the work queue, and a Search-Result message may result in sending out a Cut-Off message. Like discussed previously, the New-Work and Search-Result messages are between single processors but the Cut-Off message is a broadcast. All communication is asynchronous.

The authors describe how they implemented unique signature's for each node. The approach is common in trees. A single node's signature is the signature of its parents concatenated with a number representing which child it is of the parent. For instance, in a binary tree if the root has signature "0" the left child would be "00" and the right child would be "01." In this case, the signature of the root is a prefix for all of its children. Pruning is very easy in this case. For example, removing the subtree rooted at "0" just find all nodes with a signature that start with the prefix "0." This works for all trees. Again, this is a powerful concept that is useful even beyond game trees.

The authors also describe their strategy to prevent deadlocks. When encountering identical nodes "then TDSAB the shallower one." They mention that even if the deeper one has started searching, they still have to start a search from the shallower node. There may be repeated work but at least there won't be a deadlock, and in practice there is little additional overhead.

The authors implemented TDSAB for two games with different characteristics. The first was Awari; a game with a low branch factor (moves per board position) with rather simple evaluation. The second was Amazons, a game with a large branch factor and a more complex and expensive evaluation function. They chose these because they were being used in active research, as opposed to chess which was more historical. Small tweaks were made to make the algorithm more appropriate for the specific games.

The performance results are listed in a table at the end of the paper. They tested on clusters ranging from 1 to 64 processors. In the case of Awari, they looked 24 moves ahead and achieved a speedup of 21.82 using 64 processors. They compared their results to previous research on an implementation of checkers that had gotten a speedup of 14.35 with 64 processors. TDSAB clearly showed an improvement.

TDSAB's results were comparable to an implementation by Multigame that got a 21.54 speedup for checkers.

For Amazons they were able to get a 23.50 speedup with 64 processors. There were no fair comparisons between this implementation and most other previous research. The authors argued that measuring speedups should always be "computed relative to the best sequential algorithm." The older research implemented regular αβ algorithms, but MTD(f) has proven better. They suggested that the previous results be updated before comparing it to their version.

In conclusion they considered their results "encouraging." They state that the improved sequential algorithm MTD(f) creates some parallelization challenges that were not in αβ. They feel that they were able to successfully move TDS into the two-player domain and that there is still much to explore further.

Their results were eye opening. They were not even getting 50% efficiency but claimed that they were achieving results that were comparable with previous bests. This was an indicator as to how challenging this really is to implement.

Design of Our Software:

This provides a general description of the program and its major components. For additional technical documentation, see the source code documentation.

Source Code Summary:

Class BitUtil:
This class contains miscellaneous bit manipulation algorithms. This library is needed for the bitboard engine.

Interface OthelloBoard:
This interface is a general interface for any Othello Board. This was developed so that modules such as the GUI could be developed concurrently with the main data structure.

Class OthelloBitBoard:
This is an implementation of OthelloBoard that uses bitboards to store the board information. Functionality such as enumeration of legal moves and execution of moves is highly optimized using advanced bitboard-based techniques.

Class TableGenerator:
This class precomputes a 512kb table of the effects of any move within a single row of an OthelloBitBoard. A row bitboard is 8 bits. Since both our own and enemy pieces matter here, we need to index over both all 256 combinations of our pieces and 256 combinations of enemy pieces. Since there are 8 places to go, we have another dimension in our array. So, this table is a mapping of [0-255][0-255][0-7] to [0-255]. If

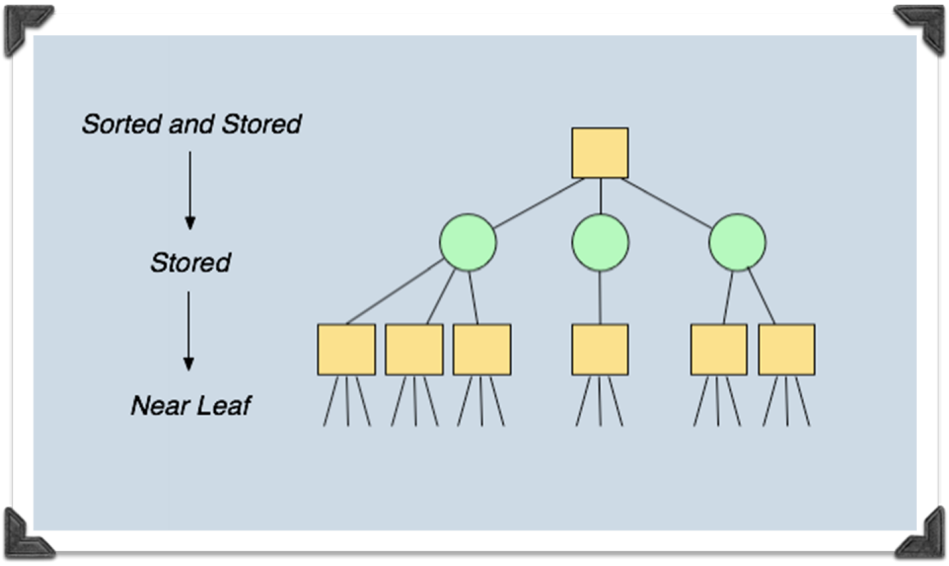
nothing changes, the move is illegal, and hence any illegal input such as overlapping friendly and enemy pieces is stored as ‘no change’.

Class Rom:
All precomputed results are accessible from this class.

Class BasicGui:
This class provides a GUI that can allow the user to play a game against the sequential iterative MTD(f) algorithm. Features are limited, as this is only for debugging and illustration.

Class OthelloAlphaBeta:
Superclass for all searches. It returns the Negamax score for the position. In it’s basic form, it is sequential. This class can perform an alpha beta search in 3 ways:
a) Barebones alpha-beta search
b) Alpha-beta with Memory (transposition table, which is a hash table)
c) Sorted Alpha-Beta with Memory

Nodes near the root benefit from ‘smart’ choices, where nodes near the leaves benefit from raw barebones performance. Additionally, storing all nodes in the transposition table would take an enormous amount of memory (gigabytes), so we do not store past a certain depth in the tree. ‘MinDepthToStore’ (see the User’s Manual) specifies how far we must be from the leaf to store in the table. Hence we arrange the search in this way, combining all 3 search functions:



This class also provides a method for “re-searching”, which is a narrow-window alpha-beta search about a known score to acquire the best legal move (as opposed to the

score of the position). By using the transposition table, the result is usually instantaneous.

Nodes are sorted based on how likely the score is to be high—hence how likely the score is to produce a cutoff. By searching these first, pruning is more aggressive but still correct.

Class OthelloMTDf:
This class inherits from OthelloAlphaBeta. This program is also sequential. It uses a series of null-window Alpha Beta searches to find the score using the MTD(f) algorithm. It also can find the score using MTD(f) in an iterative framework. Because an approximate guess can help, as well as hints in the transposition table for sorting nodes, this performs faster.

Class OthelloAlphaBetaSMP:
This class inherits from OthelloAlphaBeta. Using multiple threads, Alpha-Beta is executed in parallel.

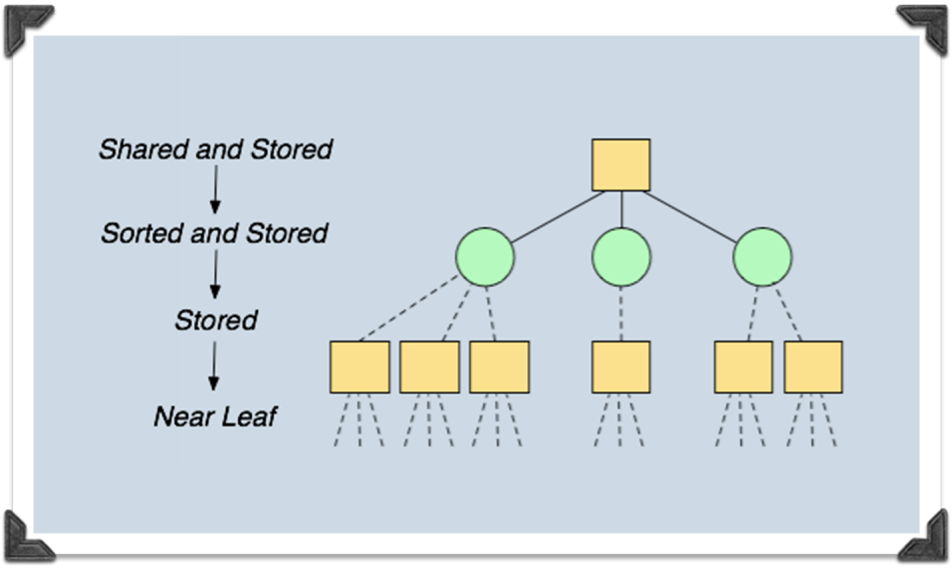
All parallel execution in our program occurs because each thread has its own queue of JobRequest objects. Any job inherits from this superclass.

For this algorithm, the uppermost levels of the tree are enumerated as AlphaBetaJobRequest objects. Threads can then execute the jobs in parallel. (The exact number of levels in which we are referring to is set by 'SharedSearchDepth'; see the User's Manual). Jobs can be stolen if there is no work for a thread.

These 'shared'/'parallel' nodes are stored in 3 data structures.
a) The job queue corresponding to the thread that plans to execute it
b) The transposition table
c) They are linked to each other, forming a sort of 'linked tree'. This is to pass information and update each other rapidly. Every time a job completes, windows may shrink and jobs may even become obsolete. Thus, this updating should be—and is—accomplished immediately.

There is a shared transposition table that is accessible to all threads. The parameter 'SharedTableLevel' (see the User's Manual) dictates how many levels in the tree are stored and read in the shared table. After this cutoff point, threads have their own, local transposition table for smaller but more rapidly accessed results. This is achieved using the SplitTranspositionTable class.

Here is a visual for the design here:



Class OthelloMTDfSMP:
This class inherits from OthelloAlphaBetaSMP. It finds the score by making use of several Parallel null-window Alpha Beta calls, according to the MTD(f) algorithm. MTD(f) requests are also jobs, called “MTDfJobRequest”.

Additionally, this search can be done iteratively. If this is done, then an IterativeMTDfJobRequest makes several MTDfJobRequest.

Developer's Manual:

On the Command Line:

This will take you through the steps to compile the packages and some required files.

1. Extract the files from the provided package. The outermost folder is called “ROOT” from here on.
2. Add the pj.jar to your classpath. (This assumes running in a bash shell):

```
$ export CLASSPATH="$CLASSPATH:/home/fac/ark/public_html/pj.jar"
```

3. Change directory into the “ROOT/src/” directory and and execute the following to compile the “core” and “gui” packages using Java 5:

```
$ alias java5c='/usr/local/versions/jdk-1.5.0_15/bin/javac'
$ java5c -d ../bin core/*.java gui/*.java
```

4. Change directory into the “ROOT/bin/” directory and generate the required RowLookup.dat file. Again, using Java 5:

```
$ alias java5='/usr/local/versions/jdk-1.5.0_15/bin/java'
$ java5 core.TableGenerator
done
```

For Eclipse:

If you have Eclipse setup with the Subclipse [http://subclipse.tigris.org/] (Subversion) plugin it may be easier to checkout our project directly into Eclipse:

1. Select File > New > Other > Checkout Projects from SVN
2. Select Create a new repository location and Next
3. Input this url: https://rit-othello.googlecode.com/svn/trunk and Click Next
4. In the resulting list for that location expand the repository and select the folder named “Othello” and Click Next.
5. Check out as a project in your workspace with an appropriate project name. Click Finish.
6. If pj.jar is not in your build path you will have to add it:
   6.1. Expand the Project
   6.2. Right click the “JRE System” entry that shows the jars in the build path. Select Build Path > Configure Build Path.
   6.3. In the “Libraries” tab click “Add External Jar” and import the Parallel Java Library.
7. This completes the setup for Eclipse.

User's Manual:

Console Execution:

This section described how to execute console driven tests designed to produce measurements. There are the four applications of interest:

OthelloAlphaBeta
OthelloAlphaBetaSMP
OthelloMTDf
OthelloMTDfSMP

The command line should look like this (using Java 5) from the “ROOT/bin” directory:

```
$ java core.OthelloAlphaBeta [InputFile]
$ java -Dpj.nt=x core.OthelloAlphaBetaSMP [InputFile]
$ java core.OthelloMTDf [InputFile]
$ java -Dpj.nt=x core.OthelloMTDfSMP [InputFile]
```

where x is the number of threads, and [InputFile] is an input file which is described in this document.

Input File Format:

The input file is a text file. It begins with any number of name/value pairings. It then must end with a “Board:” label followed by a specific text-based Othello game board. Sample input files are included; it is encouraged to view those to clarify the formatting. However, this document contains the complete list of accepted values which can be set.

Name/Value Pairings:

The program can allow specific settings to be set by adding a name/value pair on a single line per entry. For example:

```
value = 55
```

Would allow the program to map 55 to the string ‘value’.

- The only required value is ‘Turn’ is a required value; the rest are optional and have default settings.
- Case is ignored in all names and values.
- Whitespace to the right or left of either a name or a value is ignored.
- If the value is unknown to the program, then the line is ignored. This means special keywords added for only one of the algorithms will simply be ignored by the
- other algorithms. Hence special parameters do not require you to make excessive copies of a test case for each different algorithm

Recognized Settings for all programs:

Name:	Value:	Default:	Description:
Turn	white or black	**NONE**	**REQUIRED.** The player who’s turn it is.
MaxSearchDepth	Integer	12	Search depth (number of ply to look ahead)
LevelsToSort	Integer	3	Number of levels in the tree where we search in sorted order
MaxTableSize	Integer	750000	Maximum number of entries in the transposition table
MinDepthToStore	Integer	3	Minimum distance from leaf to store in Transposition Table
ShowMove	Boolean	false	Display the principal move that goes with the score we obtain
DebugOutput	Boolean	false	Display some normal-operation messages (verbose)

Additional Settings for all OthelloAlphaBeta and OthelloAlphaBetaSMP:

Name:	Value:	Default:	Description:
Alpha	Integer	0x80000001	Lower bound of the search window
Beta	Integer	0x7FFFFFFF	Upper bound of the search window

Additional Settings for all OthelloMTDf and OthelloMTDfSMP:

Name:	Value:	Default:	Description:
Guess	Integer	0	Initial guess for the MTD(f) algorithm
Iterative	Boolean	true	Whether or not to use Iterative framework

Additional Settings for all OthelloAlphaBetaSMP and OthelloMTDfSMP:

Name:	Value:	Default:	Description:
SharedSearchDepth	Integer	2	Number of levels of the search tree which are to be enumerated into jobs and can be executed in parallel. Has an enormous impact in on the number of jobs to be executed. A value (1-3) is recommended.
SharedTableLevel	Integer	9	Number of levels in the search tree in which Table is shared across threads.

Board input:

The file must contain a board. Before the board should be a line with “Board:”. Here is an example game board:

```
________
________
___BWB__
___BB___
__WWBB__
___BBW__
___B____
________
```

An underscore represents a blank space, B represents a black piece, and W represents a white piece. Note that the board is 8*8 and hence requires 8 characters on each of 8 lines. No spaces.

Output:

Upon completion, a variety of statistics will be printed.

Name:	Description:
Score	The score of the position. Higher means better for the current player
leaf nodes	Number of leaf nodes evaluated
non-leaf nodes:	Number of non-leaf nodes evaluated
Leaf nodes/sec	Rate in which leaf nodes were processed
nodes retrieved	Number of Transposition Table lookups (it counts only when the table actually had the position that the algorithm was looking for)
table size:	Number of entries in the transposition table
Search time:	Total search time

Additional stats are printed for SMP versions:

Name:	Description:
totalJobsExecuted	Number of Jobs that were executed (Jobs are executed in Parallel)
leafJobsExecuted	Number of Jobs that invoked Sequential searches, rather than being split into child jobs
jobsSkipped	Number of jobs that were requested, but then not executed.

Additionally, this will be output if ShowMove=true is added to the input file:

Name:	Description:
BestMove	the (x,y) pair of coordinates in which the current player should place their piece
re-search time	Time spent acquiring the best move based on the score

GUI Execution:

The GUI can be run by executing BasicGui. There are no command line arguments.

The basic GUI allows you to play as black with the first move. The AI plays white, using the sequential iterative-deepening MTD(f) search. You cannot benchmark using the GUI program; it is merely a demonstration of the AI's playing capabilities.

You can play a complete game using the GUI. It is highly competent and fun to play against. Only by taking definitive advantage of its early-in-the-game strategic mistakes can you possibly win.

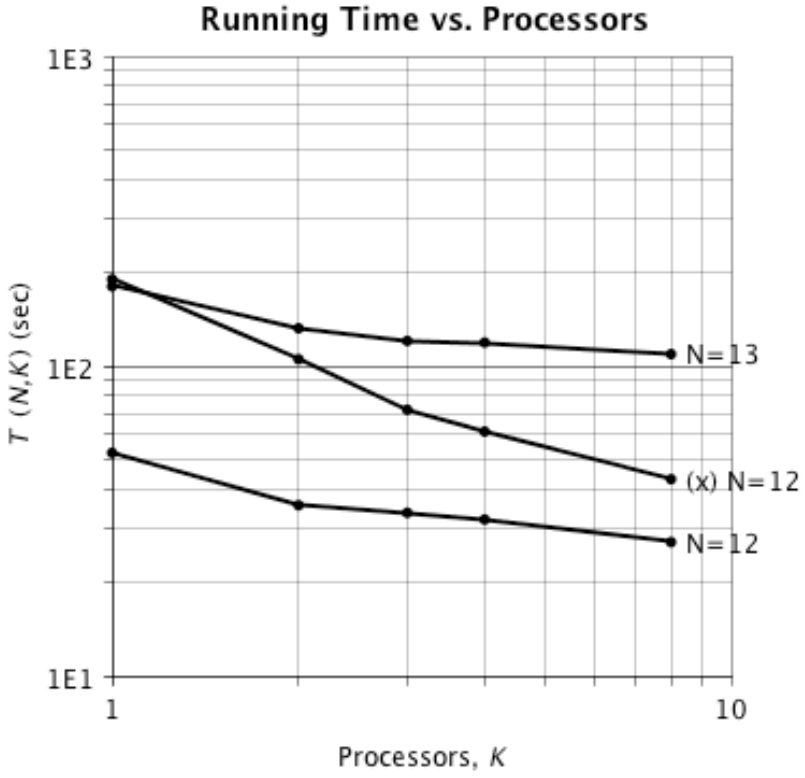
Note: Should you have NO legal moves, the AI may act again instantly. This may be confusing if you don't know that this can happen.

Performance Metrics:

We were able to run performance metrics on our Alpha-Beta SMP implementation in a few stages. Early on we ran some tests on our first relatively stable implementation of our algorithm. This version showed some very promising parallel speed-ups. At 8 processors it got a speedup of 4.4. This was roughly comparable to the speedup of 4.6 that one of the papers showed their algorithm got at the same number of processors.

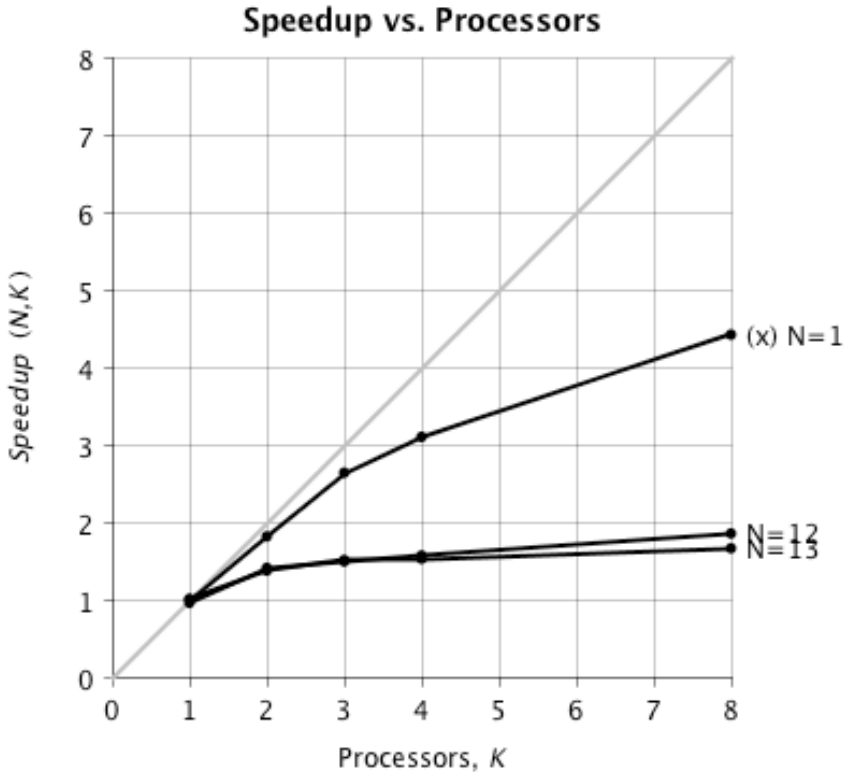
However, as we continued to improve our SMP version we were able to make significant improvements on the overall runtime by moving from a single work queue for all threads to an individual work queue per thread. Although this had a global improvement, the parallel performance dropped. Also, the stability of the algorithm improved. Previously we had seen dramatic changes in run-times in the older algorithm, now we were seeing more consistent results.

In these graphs N is the lookahead depth we ran the algorithm on. We ran multiple tests for K (the number of threads) = 1, 2, 3, 4, and 8. The line labeled “(x)” was the older algorithm before the overall speedup:

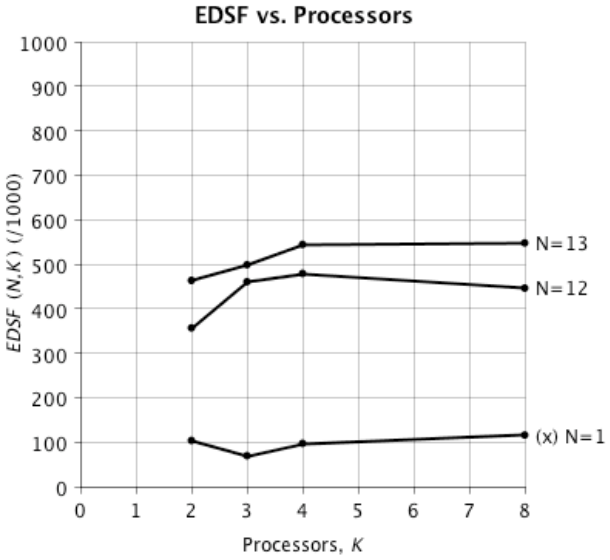


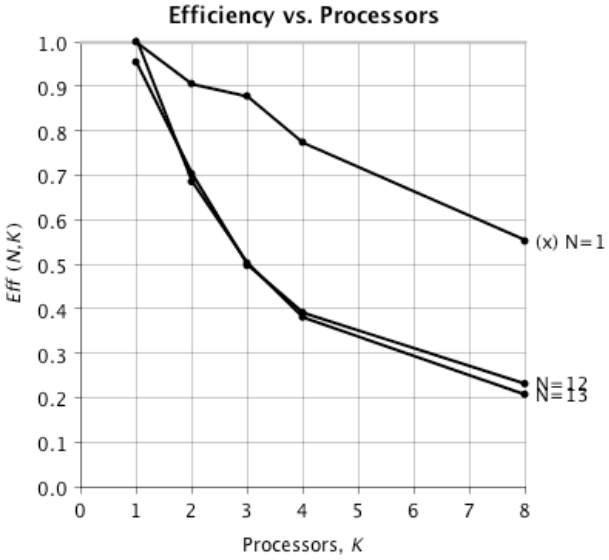
Here you can see that moving from the older algorithm to the newer algorithm we were able to increase the size of the problem and achieve better times, however we suffered from worse parallel speed-ups. We feel this is because although our second

implementation was fundamentally better we didn't properly load balance it to give all the processors enough work. Experimenting with some of the parameters, and the load



balancing system would eventually have produced better performance.
Again, the same explanation applies for the speedup graph. We feel that if we could get the load balancing correct for the newer implementation we would see the same kind of

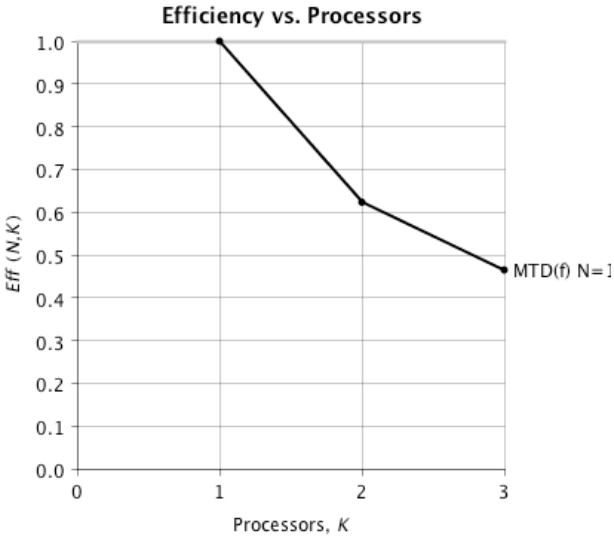


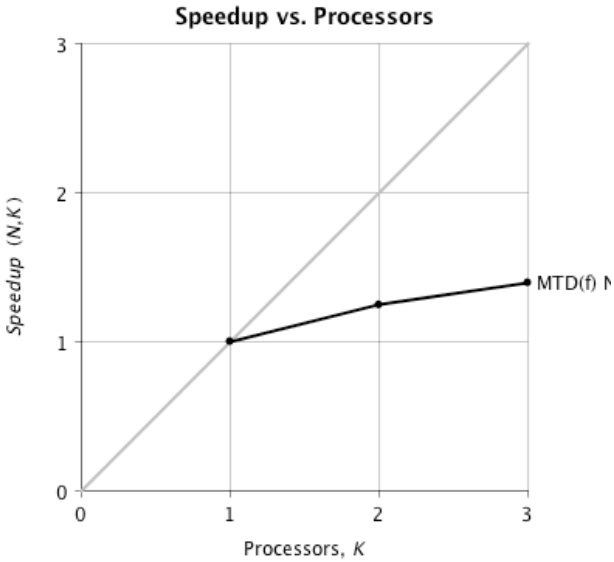


speedup that we saw for the older algorithm.

Apart from run-time performance we know for certain the the parallel implementation was indeed evaluating more nodes per second then the sequential version. We kept a few counters as indicators of the number of nodes that were being evaluated. Although not synchronized they still indicated that the parallel version was doing more work per unit of time then the sequential. Thus, the parallel version did show improvements on the sheer amount of processing. That means that in some cases, efficient cut-offs arrived at earlier due to parallel processing in the parallel implementation could provide a large speed-up.

The results for our SMP implementation of MTD(f) were the same. Unfortunately when we ran these metrics as the number of threads increased the table size grew too quickly for lookaheads of the same depth that we tested in our Alpha-Beta algorithm. Here are our results on the input set of comparable size to the above Alpha-Beta test from 1, 2, and 3 threads. The problem has been fixed but the results were about the same:





Here are the raw results. (m) in this table is the MTD(f) implementation:

N	K	T	Spdup	Effic	EDSF	Devi
(x)	1	190302	1.000	1.000		15%
(x)	2	105037	1.812	0.906	0.104	15%
(x)	3	72240	2.634	0.878	0.069	19%
(x)	4	61446	3.097	0.774	0.097	68%
(x)	8	43071	4.418	0.552	0.116	67%

N	K	T	Spdup	Effic	EDSF	Devi
12	seq	50016				13%
12	1	52400	0.955	0.955		0%
12	2	35543	1.407	0.704	0.357	2%
12	3	33555	1.491	0.497	0.461	6%
12	4	31926	1.567	0.392	0.479	9%
12	8	27010	1.852	0.231	0.446	15%
13	seq	181390				0%
13	1	180756	1.004	1.004		1%
13	2	132254	1.372	0.686	0.463	1%
13	3	120348	1.507	0.502	0.499	8%
13	4	118974	1.525	0.381	0.544	2%
13	8	109242	1.660	0.208	0.548	36%
(m)	seq	82041				4%
(m)	1	90275	0.909	0.909		1%
(m)	2	72435	1.133	0.566	0.605	12%
(m)	3	64728	1.267	0.422	0.576	31%

What We Learned:

Perhaps the biggest lesson is that there is a vast gray area between massively parallel and purely sequential problems. Many of the examples demonstrated in class are parallelized simply by breaking up a single large for loop, possibly with the addition of load balancing. Anything other than the slight amount of communication necessary for load balancing seems to have a disproportionate effect on the difficulty of implementing an efficient algorithm. Multiple processes doing unpredictable amounts of work naturally tend to have complicated behavior. This in turn requires a shift in the programmer's thought processes, in a way that parallelizing the Delaunay triangulation did not.

Another lesson comes from our performance metrics. A gain in absolute performance may produce a decrease in parallel speedup. Our refined algorithm performs better on one node than our original test algorithm did on four nodes. However, the original algorithm exhibited far superior parallel efficiency. In the case of game tree search, it seems that our parallel performance was inversely proportionate to our distance from a brute force search. This makes sense; as we are able to eliminate more and more of the search tree, our synchronization needs increase, and we change our minds about which parts need to be evaluated.

Possible Future Work:

Improving our strategy algorithm could be another such step in the right direction. We might be able to quickly fill in standard known best openings and then calculate the results of those moves. Using these techniques we might be able to find the optimal moves more quickly or search deeper then would otherwise be possible. Chess programs sometimes use a table of pre-computed openings.

As previously mentioned, our improved algorithm shows better overall performance but a much higher EDSF. Reducing this would not only speed up our SMP version but might also aid us in developing a cluster or cluster hybrid version. Load balancing would be critical, and message passing would become more costly, but the potential for far higher numbers of processes might make the effort worthwhile.

Unlike chess or checkers, Othello's starting position exhibits 180 degree rotational symmetry. It's possible that the future moves in the game tree may also exhibit rotational symmetry. If this is taken into account, it might be possible to optimize game tree searches, particularly in calculating openings.

Tweaked job stealing might well be what is needed to push this program closer to the ideal speedup. Because of the way threads can become specialized, jobs may be evaluated significantly slower in the “wrong” thread than in the ideal one. (We have seen results up to 30x slower.) Under normal circumstances, duplication of work in a parallel program is something to avoid whenever possible. However, it may be beneficial to allow threads to steal each other's jobs under certain circumstances. Determining these circumstances and developing the corresponding heuristics and rules is the challenging part.

Another interesting challenge would be tackling the game of Go. While even the best Go programs can't compete with skilled human players, we could see how our program stacks up against some others. Additionally, it would be interesting to see whether the increased branching factor of the game tree allows us to increase our speedup across multiple CPUs, as a higher number of branches might allow more processes to stay busy with useful work.

References:

Kishimoto, A. and Schaeffer, J. 2002. Distributed Game-Tree Search Using Transposition Table Driven Work Scheduling. In Proceedings of the 2002 international Conference on Parallel Processing (Icpp'02) (August 18 - 21, 2002). ICPP. IEEE Computer Society, Washington, DC, 323.
http://ieeexplore.ieee.org.ezproxy.rit.edu/stamp/stamp.jsp?arnumber=1040888&isnumber=22315

Borovska, P. and Lazarova, M. 2007. Efficiency of parallel minimax algorithm for game tree search. In Proceedings of the 2007 international Conference on Computer Systems and Technologies (Bulgaria, June 14 - 15, 2007). B. Rachev, A. Smrikarov, and D. Dimov, Eds. CompSysTech '07, vol. 285. ACM, New York, NY, 1-6.
http://delivery.acm.org.ezproxy.rit.edu/10.1145/1340000/1330615/a14-borovska.pdf?key1=1330615&key2=1390949221&coll=Portal&dl=GUIDE&CFID=14888790&CFTOKEN=20634058

Plaat, A., Shaeffer, J., Pijls, W., and de Bruin, A. 1996. Best-first fixed-depth minimax algorithms. Artif. Intell. 87, 1-2 (Nov. 1996), 255-293.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.8227&rep=rep1&type=pdf

Aske Plaat, "Research Re: search & Re-search," PhD Thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, Thesis Publishers, Amsterdam, The Netherlands, June 20, 1996.
http://www.cs.vu.nl/~aske/Papers/abstr-ks.html