Report Common AssignmentOpenMP

Alpha-beta pruning algorithm

Lecturer: Francesco Moscato - **fmoscato@unisa.it**

Group:

- Milone Marco 0622701992 m.milone15@studenti.unisa.it
- Sarno Fabrizio 0622702052 f.sarno14@studenti.unisa.it
- Squitieri Beniamino 0622702021 b.squitieri@studenti.unisa.it

Summary

Problem description

Provide a parallel version of the Alpha-beta pruning algorithm to improve MinMax algorithm's performances in Tic-Tac-Toe game. The implementation MUST use shared memory paradigm and has to be implemented by using OpenMP. Students MUST provide parallel processes on different game tree's branches, and each process has to evaluate the best move to make in order to win the match against the other opponent.

Solution

The partitioning of the tree for parallel computation is done on a per-child basis. Each child of the tree evaluates the minimum and maximum moves together without depending on the results obtained from other children and thus does all the move computation relatively fast. After this computation is done, the main thread collects the final result and returns the final results.

The program consists in a player (marked with *X* character) against an AI (marked with *O* character). Player's moves have been calculated executing the same program with two AIs in order to obtain the correct pattern of moves so that the match will always end in a tie. That's because both of them use the alpha-beta pruning algorithm to compute their next best move and minimize the opponent's one. However this solution has not been studied and reported in the following document since this would have meant doing the same algorithm's evaluation twice.

After that, every *X*'s move has inserted in an array of pairs. Then, for each turn, the pair at index *i* in the array has been picked and drew on the board.

Theoretical hints
Alpha-beta pruning algorithm

Alpha-beta pruning is a search algorithm commonly used in game trees and decision trees to reduce the number of nodes that need to be evaluated. This technique allows for the efficient search of large decision spaces by avoiding the evaluation of branches that are unlikely to lead to a better result.

The algorithm works by maintaining two values for each player: alpha and beta. Alpha represents the maximum value that the current player can achieve, and beta represents the minimum value that the opponent can achieve. As the search progresses, if the algorithm finds a node that exceeds the current alpha or beta value, it stops searching further in that direction because it is unlikely to lead to a better result.

This technique is especially useful for machine playing of two-player games like Tic-tac-toe and especially for games like chess, checkers, or Connect 4, where the decision tree can be quite large. By pruning branches that are unlikely to lead to a better outcome, alpha-beta pruning can significantly reduce the number of nodes that need to be evaluated and improve the efficiency of the search algorithm.

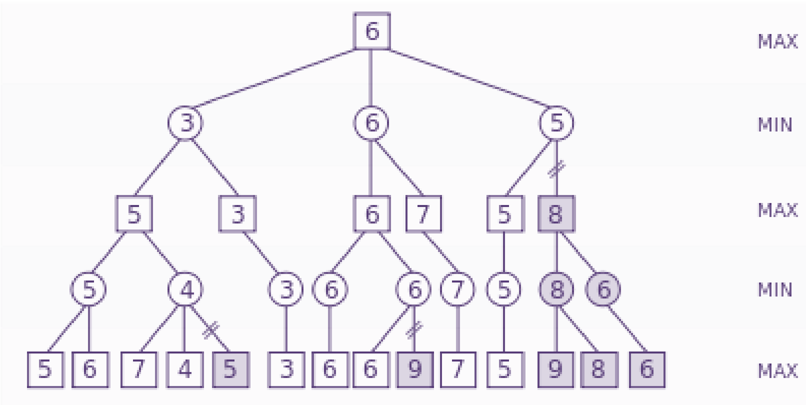


Figure 1 - An illustration of alpha–beta pruning. The dashed branches don't need to be explored.

OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran on most platforms, instruction set architectures, and operating systems, including Unix and Microsoft Windows platforms. It is a low-level API that supports parallel processing in applications by allowing them to be executed on multiple processors, cores, or threads in a shared memory environment. OpenMP provides a simple and flexible model for parallel computing, making it easy for programmers to parallelize existing code, reducing development time and cost. It is widely used in scientific and engineering applications, high-performance computing, and data analytics. OpenMP is an open-source project and is maintained by the OpenMP Architecture Review Board (ARB), which includes representatives from hardware vendors, software vendors, and academic institutions.

Figure 2 - OpenMP logo

Experimental setup
Hardware
CPU

Processor	8
Vendor_id	GenuineIntel
CPU family	6
Model	142
Model name	Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
Stepping	12
Microcode	0xffffffff
CPU Mhz	2304.007
Cache size	8192 KB
Physical ID	0
Siblings	8
Core ID	3
CPU cores	4
Apicid	7
Initial apicid	7
Fpu	Yes
Fpu_exception	Yes
Cpuid level	22
wp	Yes
flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon rep_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pdcm pcid sse4_1 sse4_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow vnmi ept vpid ept_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush_l1d arch_capabilities
vmx flags	vnmi invvpid ept_x_only ept_ad ept_1gb tsc_offset vtpr ept vpid unrestricted_guest ept_mode_based_exec
bugs	spectre_v1 spectre_v2 spec_store_bypass swapgs itlb_multihit srbds mmio_stale_data retbleed eibrs_pbrsb
bogomips	4608.01
clflush size	64
cache_alignment	64
address sizes	39 bits physical, 48 bits virtual

RAM

MemTotal	8026084 kB
MemFree	7356764 kB
MemAvailable	7338972 kB
Buffers	11276 kB
Cached	177900 kB
SwapCached	0 kB
Active	59828 kB
Inactive	300392 kB
Active(anon)	1928 kB
Inactive(anon)	171632 kB
Active(file)	57900 kB
Inactive(file)	128760 kB
Unevictable	0 kB
Mlocked	0 kB
SwapTotal	2097152 kB
SwapFree	2097152 kB
Dirty	0 kB
Writeback	0 kB
AnonPages	171204 kB
Mapped	96060 kB
Shmem	2508 kB
KReclaimable	22868 kB
Slab	66784 kB
SReclaimable	22868 kB
SUnreclaim	43916 kB
KernelStack	3712 kB
PageTables	6868 kB
NFS_Unstable	0 kB
Bounce	0 kB
WritebackTmp	0 kB
CommitLimit	6110192 kB
Committed_AS	790984 kB
VmallocTotal	34359738367 kB
VmallocUsed	23892 kB
VmallocChunk	0 kB
Percpu	2176 kB
AnonHugePages	51200 kB
ShmemHugePages	0 kB
ShmemPmdMapped	0 kB
FileHugePages	0 kB
FilePmdMapped	0 kB
HugePages_Total	0
HugePages_Free	0
HugePages_Rsvd	0
HugePages_Surp	0
Hugepagesize	2048 kB
Hugetlb	0 kB
DirectMap4k	21504 kB
DirectMap2M	4061184 kB
DirectMap1G	13631488 kB

Software

- Ubuntu 20.04.1
- GCC 9.4.0

Performance, Speedup & Efficiency

Each case study represents a level of compiler optimization (-O0, -O1, -O2, -O3), therefore, both parallel and sequential versions will be compiled with the corresponding optimization.

O0: This is the lowest level of optimization, where the compiler produces the simplest and most readable executable code possible. The resulting code is easy to debug, but may be slower than other optimization levels. At this level, the compiler does not do much advanced optimization, and so the resulting code may be less efficient in terms of execution time and memory usage.

O1: Optimization level O1 is a small optimization that focuses mainly on reducing repetitive computation operations. The compiler performs some simple optimizations, such as reducing the number of instructions for a given operation and using registers to store common values. This level of optimization is quite safe, and the resulting code should be faster than the code generated with the O0 option.

O2: Optimization level O2 is a more advanced version of level O1. Here, the compiler performs a wide range of advanced optimizations, such as parallelizing operations, eliminating unnecessary code, using registers to store temporary values, and replacing expensive operations with faster ones. The end result will be faster code, but it may be slightly more difficult to debug.

O3: The O3 optimization level is the maximum optimization available in the compiler. Here, the compiler performs all available optimizations, including advanced data flow analysis, parallelization of code, use of low-level instructions, replacement of expensive operations with faster ones, and elimination of all unnecessary code. The end result should be highly optimized and fast code, but it may be very difficult to debug and may require more memory resources.

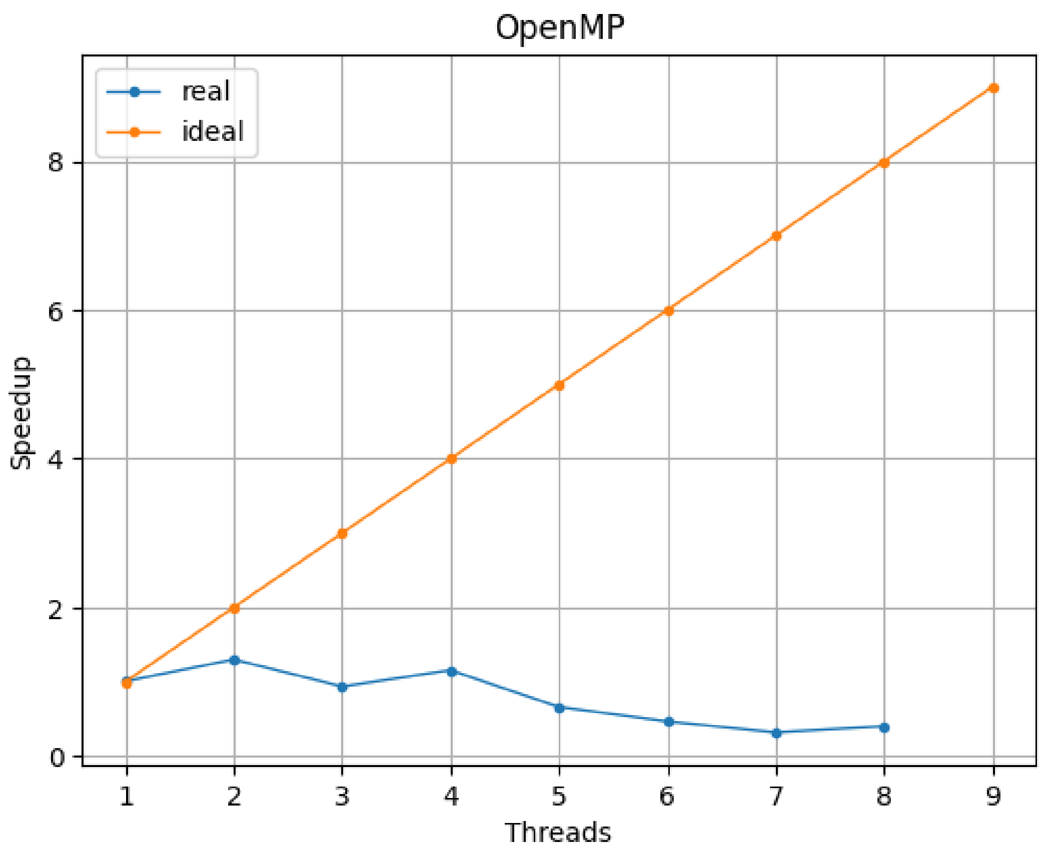
In order to see how much our parallelization has been effective, the speedup has been evaluated: Speedup is a measure that allows to evaluate a developed parallel algorithm, specifically it is given by the ratio of the execution times of the sequential version to those of the parallelized version. Speedup values greater than 1 indicate that parallelization has provided benefits, while values less than 1 indicate that perhaps communication times between processes may have impacted more than the benefits of running code in parallel.

Next the final results will be presented in a plot, in which the ones obtained by parallelization will be compared with the ideal case in which the whole algorithm was perfectly parallelizable, since the speedup depends precisely on the sequential part of the problem and the parallel part, and by increasing the number of threads we are essentially going to work on the latter.

Optimization 0

About this optimization, the results obtained show that an execution parallelized between two threads is the best alternative. In general, until the fourth thread the speedup values are optimal. Whereas from the fifth thread they are unused because the space state will reduce for each move done during the game, so there will be more threads than branches to assign. Because of it, threads will be allocated but unused, so there will be no more advantages for computation's time.

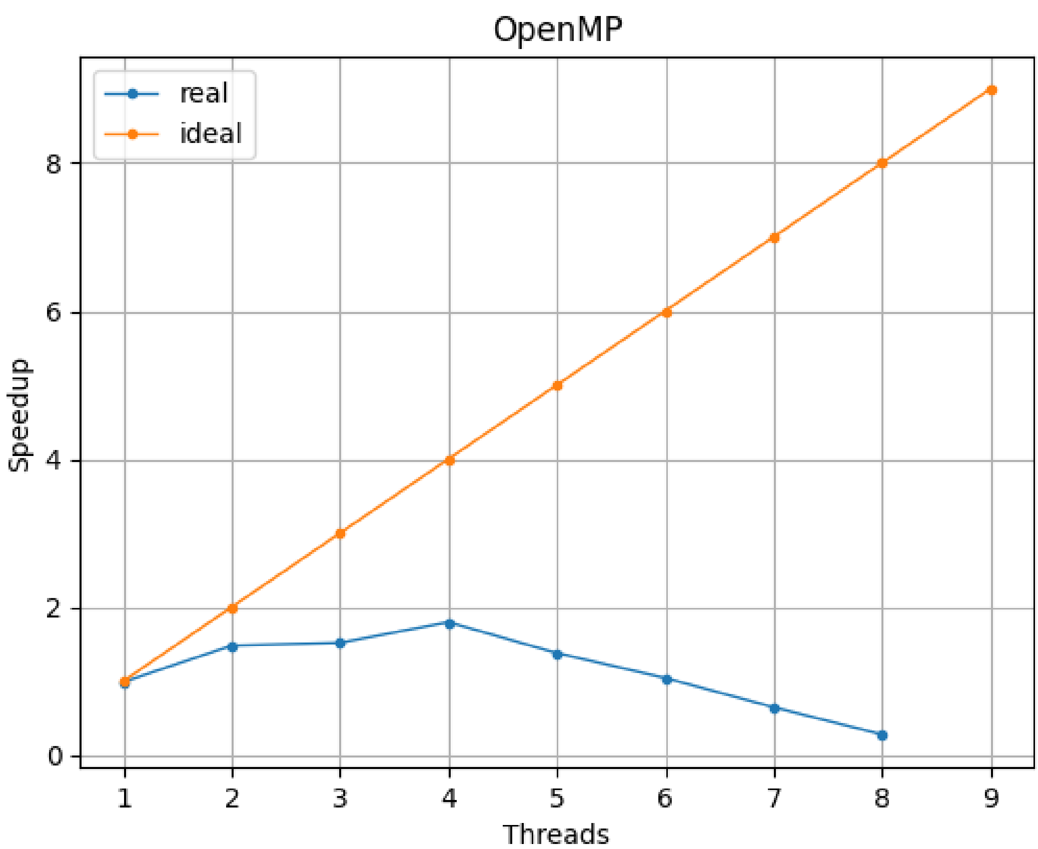
	Threads	Time(s)	Speedup
Sequential	0	0.0085539	1
OpenMP	1	0.0085053	1.0057059
OpenMP	2	0.0065821	1.2995686
OpenMP	3	0.0091364	0.9362460
OpenMP	4	0.0074074	1.1547765
OpenMP	5	0.0129370	0.6611930
OpenMP	6	0.0183371	0.4664785
OpenMP	7	0.0268288	0.3188315
OpenMP	8	0.0212904	0.4017704



Optimization 1

About this optimization, it is equal or slightly different than the previous one: the different optimization type lead to results that change by few digits.

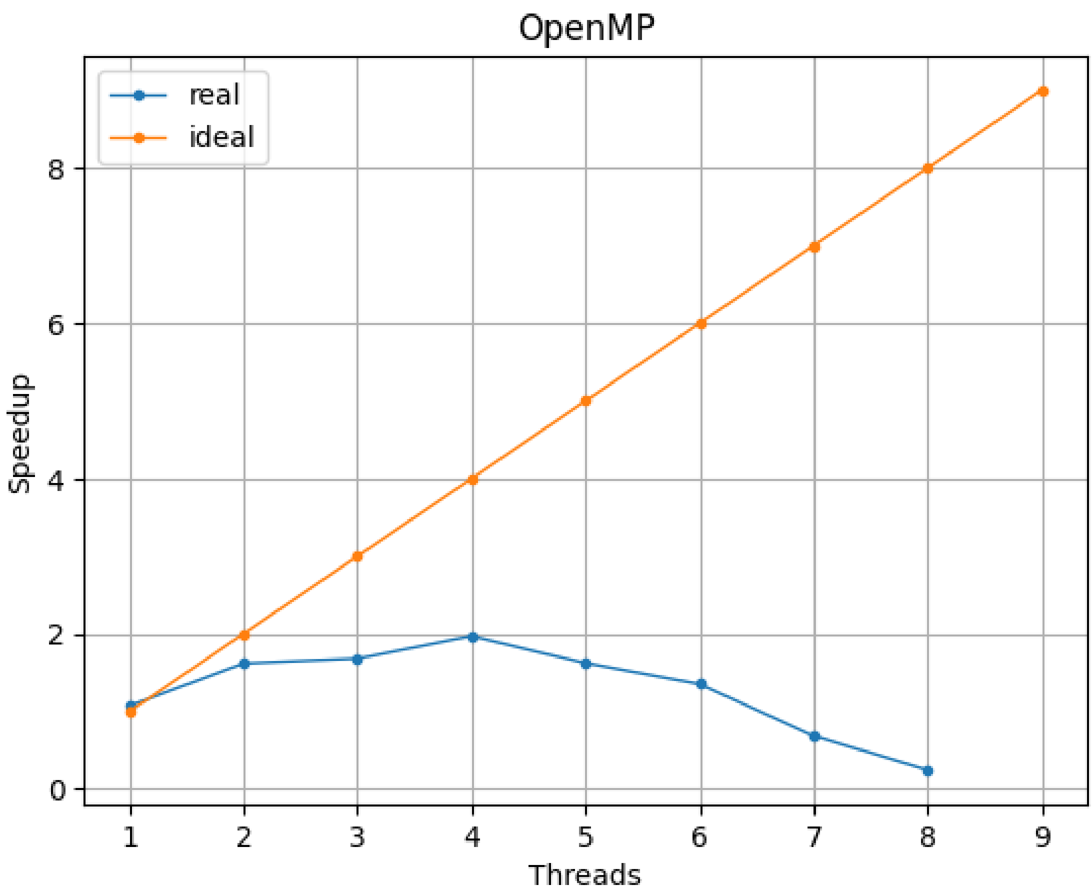
	Threads	Time(s)	Speedup
Sequential	0	0.0050908	1
OpenMP	1	0.0051507	0.9883676
OpenMP	2	0.0034423	1.4789119
OpenMP	3	0.0033558	1.5170323
OpenMP	4	0.0028360	1.7950548
OpenMP	5	0.0036938	1.3782116
OpenMP	6	0.0048839	1.0423732
OpenMP	7	0.0078724	0.6466660
OpenMP	8	0.0182422	0.2790682



Optimization 2

About this optimization, it is equal or slightly different than the previous one: the different optimization type lead to results that change by few digits.

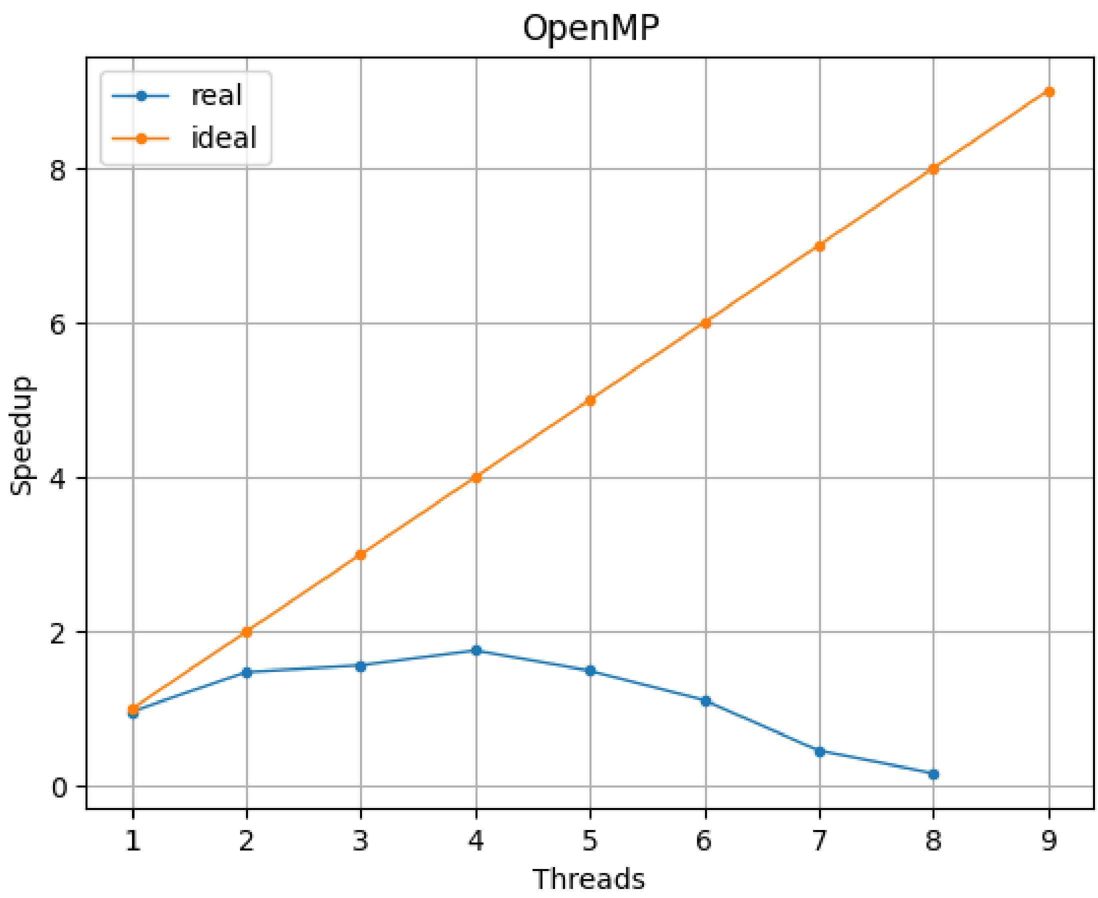
	Threads	Time(s)	Speedup
Sequential	0	0.0048573	1
OpenMP	1	0.0045165	1.0754642
OpenMP	2	0.0030029	1.6175509
OpenMP	3	0.0028882	1.6817802
OpenMP	4	0.0024668	1.9690341
OpenMP	5	0.0029933	1.6227112
OpenMP	6	0.0035880	1.3537624
OpenMP	7	0.0070749	0.6865557
OpenMP	8	0.0195447	0.2485218



Optimization 3

About this optimization, it is equal or slightly different than the previous one: the different optimization type lead to results that change by few digits.

	Threads	Time(s)	Speedup
Sequential	0	0.0033457	1
OpenMP	1	0.0034668	0.9650632
OpenMP	2	0.0022637	1.4779947
OpenMP	3	0.0021391	1.5641017
OpenMP	4	0.0019042	1.7569936
OpenMP	5	0.0022383	1.4947840
OpenMP	6	0.0030051	1.1133273
OpenMP	7	0.0072406	0.4620762
OpenMP	8	0.0198552	0.1685055



Considerations

From the analysis carried out, it is agreed that the parallel version of the algorithm is slightly better than the sequential one. However, the advantages are not so remarkable considering the game chosen. Indeed, Tic-tac-toe has a little space state and also very few moves possible. So there is not a deep change in time execution and used resources between the sequential and parallel versions. Using parallelized alpha-beta pruning on games like checkers, chess or others more complex than Tic-tac-toe will highlight better the differences with the sequential version and also improve the resources management reducing at the same time the execution time.

Source Code

For Alpha-beta pruning sequential algorithm, reference was made to an open source code available on GitHub at the following link: https://github.com/GeorgeSeif/Tic-Tac-Toe-AI

Other references for alpha-beta pruning algorithm have been made to the following documents:

- *Comparative study of performance of parallel Alpha-Beta Pruning for different architectures* – S. P. Singhal, M. Sridevi;
- *PARALLELIZATION OF ALPHA-BETA PRUNING ALGORITHM FOR ENHANCING THE TWO PLAYER GAMES* – A. Kumari, S. Singh, S. Dalmia, V. Geetha.

For doubts and difficulties encountered with OpenMP, reference was made to its official online documentation available at the following link: https://www.openmp.org/

API

Public Docs is available at the following link: https://wild-oven.surge.sh

Comparison Functions

```
int     max(int a, int b)
```

Function that finds the maximum of two integers.

```
int     min(int a, int b)
```

Function that finds the minimum of two integers

Print Functions

void **print_game_state**(**int** state)

Function that, given the status, determines whether the game ended with a victory or a defeat for the player or with a draw.

void **print_board**(**char** board[BOARD_SIZE][BOARD_SIZE])

Function that prints the Tic-tac-toe board.

Getter Functions

int **get_legal_moves**(**char** board[BOARD_SIZE][BOARD_SIZE], **struct Pair** *legal_moves)

Function that calculates all available legal moves on the board (spaces that are not occupied by either player)

int **get_occupied_positions**(**char** board[BOARD_SIZE][BOARD_SIZE], **char** marker, **struct Pair** *occupied_positions)

Function that calculate all board positions occupied by a specific marker.

char **get_opponent_marker**(**char** marker)

Function that, given a marker, returns the other player’s marker.

int **get_board_state**(**char** board[BOARD_SIZE][BOARD_SIZE], **char** marker)

Function that determines the state of a Tic-tac-toe board for a given player marker.

Check Functions

```
bool    board_is_full(char board[BOARD_SIZE][BOARD_SIZE])

                  Function that checks if the board is full
                  (no more available moves).

bool    game_is_won(struct Pair *occupied_positions, int
        occupied_positions_count)

                  Function that checks if the game is
                  won.

bool    game_is_done(char board[BOARD_SIZE][BOARD_SIZE])

                  Function that checks if the game is over.
```

Sequential MinMax functions

struct Pair **minimax_optimization**(**char** (*board)[BOARD_SIZE], **char** marker, **int** depth, **int** alpha, **int** beta)

Function that uses alpha-beta pruning algorithm to calculate the best move for the AI.

Parallel MinMax functions

struct Pair **parallel_minimax_optimization**(**char** (*board)[BOARD_SIZE], **char** marker, **int** depth, **int** alpha, **int** beta)

Function that uses alpha-beta pruning algorithm to calculate the best move for the AI player by exploiting a branch parallelization using OpenMP.

Alpha-beta Pruning functions

struct Move **alpha_beta**(**char** board[BOARD_SIZE][BOARD_SIZE], **char** marker, **int** depth, **int** alpha, **int** beta)

Function that implements the alpha-beta pruning algorithm.

Data Structures Documentation

```
struct Pair(int a, int b)
/** @brief Structure that models a pair of board coordinates
 * @param x is the value for the row
 * @param y is the value for the column*/

struct Move(int a, int b)
/** @brief Structure that models a move of the board
 * @param score is an integer indicating the quality of the move
 * @param p is a pair of coordinates of type Pair*/
```

Winning States Documentation

```
struct Pair winning_states[WINNING_STATES_SIZE][BOARD_SIZE]
/** @brief Matrix of type Pair which contains all the set of three pairs
 * representing the winning combinations of Tic-tac-toe*/
```

Comparison Functions Documentation

int max(int a, int b)

```
/** @brief Function that finds the maximum of two integers
 * @param a is the first integer
 * @param b is the second integer
 * @return returns the maximum between a and b*/
```

int min(int a, int b)

```
/** @brief Function that finds the minimum of two integers
 * @param a is the first integer
 * @param b is the second integer
 * @return returns the minimum between a and b*/
```

Print Functions Documentation

void print_game_state(int state)

```
/** @brief Function that, given the status, determines whether the game ended
with a victory or a defeat for the player or with a draw
 * @param state is an integer used to evaluate the outcome of the game*/
```

void print_board(char board[BOARD_SIZE][BOARD_SIZE])

```
/** @brief Function that prints the Tic-tac-toe board
 * @param board is matrix of char representing the Tic-tac-toe board*/
```

Getter Functions Documentation

int get_legal_moves(char board[BOARD_SIZE][BOARD_SIZE], struct Pair *legal_moves)

```
/** @brief Function that calculates all available legal moves on the board
(spaces that are not occupied by either player)
 * @param board is matrix of char representing the Tic-tac-toe board
 * @param legal_moves is a pointer of type Pair used to save all legal moves
 * @return number of legal moves*/
```

int get_occupied_positions(char board[BOARD_SIZE][BOARD_SIZE], char marker, struct Pair *occupied_positions)

```
/** @brief Function that calculate all board positions occupied by a specific
marker.
 * @param board is matrix of char representing the Tic-tac-toe board
 * @param occupied_positions is a pointer of type Pair used to save all
occupied positions
 * @param marker is a char representing the player marker
 * @return number of occupied positions*/
```

char get_opponent_marker(char marker)

```
/** @brief Function that, given a marker, returns the other player's marker
 * @param marker marker is a char representing the player marker
 * @return other player's marker*/
```

int get_board_state(char board [BOARD_SIZE][BOARD_SIZE], char marker)

```
/** @brief Function that determines the state of a tic-tac-toe board for a
given player marker.
* @param board is matrix of char representing the Tic-tac-toe board .
* @param marker is a char representing the player marker.
* @return an integer representing the state of the board for the player marker:
WIN if the player has won, LOSS if the opponent has won, DRAW if the board is
full but no one has won.*/
```

Check Functions Documentation

```
bool board_is_full(char board[BOARD_SIZE][BOARD_SIZE])
/** @brief Function that checks if the board is full (no more available moves).
 * @param board is matrix of char representing the Tic-tac-toe board
 * @return true if the board is full false otherwise*/

bool game_is_won(struct Pair *occupied_positions, int occupied_positions_count)
/** @brief Function that checks if the game is won
 * @param occupied_positions is a pointer of type Pair which contains all pairs
of occupied positions
 * @param occupied_positions_count is an integer indicating the number of
occupied cells
 * @return true if the game is won false otherwise*/

bool game_is_done(char board[BOARD_SIZE][BOARD_SIZE])
/** @brief Function that checks if the game is over.
 * @param board is matrix of char representing the Tic-tac-toe board .
 * @return true if the game is over, false otherwise.*/
```

Sequential MinMax Functions Documentation

```
struct Pair minimax_optimization(char (*board)BOARD_SIZE], char marker, int
depth, int alpha, int beta)
/** @brief Function that uses the alpha-beta pruning algorithm to calculate the
best move for the AI.
 * @param board is matrix of char representing the Tic-tac-toe board .
 * @param marker is a char representing the player marker.
 * @param depth The current depth of the search in the game tree.
 * @param alpha The best score that the maximizing player (AI player) can
guarantee at this point.
 * @param beta The best score that the minimizing player (Player) can guarantee
at this point.
 * @return a Pair representing the best move to make.*/
```

Parallel MinMax Functions Documentation

```
struct Pair minimax_optimization(char (*board)BOARD_SIZE], char marker, int
depth, int alpha, int beta)
/** @brief Function that implements the alpha-beta pruning algorithm to
calculate the best move for the AI
 * player by exploiting a branch parallelization using OpenMP.
 * @param board is matrix of char representing the Tic-tac-toe board.
 * @param marker is a char representing the player marker.
 * @param depth The current depth of the search in the game tree.
 * @param alpha The best score that the maximizing player (AI player) can
guarantee at this point.
 * @param beta The best score that the minimizing player (Player) can guarantee
at this point.
 * @return a Pair representing the best move to make.*/
```

Alpha-beta Pruning Functions Documentation

```
struct Move alpha_beta(char [BOARD_SIZE][BOARD_SIZE], char marker, int depth,
int alpha, int beta)
/** @brief Function that implements the alpa-beta pruning algorithm.
 * @param board is matrix of char representing the Tic-tac-toe board.
 * @param marker is a char representing the player marker.
 * @param depth The current depth of the search in the game tree.
 * @param alpha The best score that the maximizing player (AI player) can
guarantee at this point.
 * @param beta The best score that the minimizing player (Player) can guarantee
at this point.
 * @return A struct Move representing the best move and its corresponding
score.*/
```

How to run

Assumption: The code for generating directories, tables and plots requires the python interpreter and the *matplotlib* library to be installed entering the command: **`pip3 install matplotlib`**

1. Navigate to the folder containing the makefile
2. To clear previously obtained achievements and builds, enter the command: **`make clean`**
3. To generate the necessary directories and compile and linking the various source codes, enter the command: **`make all`**
4. To run the algorithm for making tests, producing results, measurements, graphs and tables, enter the command: **`make test`**

The results of the algorithm can be viewed in the *Results* folder, divided by type of optimization.
The execution times of the algorithm and their average values can be viewed respectively in the *Informations* and *Results* folders, divided by type of optimization.
The results in graphical and tabular format can be viewed respectively in the *Plots* and *Tables* folders, divided by type of optimization.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.