Reversi-Parallel Programming Solution

Parallel and Concurrent Programming

Moloney C M (FCES)

2023

Table of Contents

[Introduction 2](#_Toc151759198)

[Problem statement 2](#_Toc151759199)

[Logic and workings of Reversi 2](#_Toc151759200)

[How well my program works 2](#_Toc151759201)

[Analysis and future improvements 3](#_Toc151759202)

[Compare shared memory and distributed memory models in parallel programming 4](#_Toc151759203)

[Human Factors involved within the use of the system 4](#_Toc151759204)

[Appendices: 5](#_Toc151759205)

[Appendix 1 - Code 5](#_Toc151759206)

[Mailbox.h 5](#_Toc151759207)

[Mailbox.c 5](#_Toc151759208)

[Paro64bit.c 7](#_Toc151759209)

[Appendix 2 - Video Link 8](#_Toc151759210)

[Figures 8](#_Toc151759211)

[Figure 1 8](#_Toc151759212)

[Figure 2 8](#_Toc151759213)

[Figure 3 8](#_Toc151759214)

[Figure 4 9](#_Toc151759215)

## Introduction

This report provides an overview of the challenges posed in transitioning a sequential search game to a parallel search format. It delves into the mechanics of Reversi (Othello), outlining its gameplay and tactical strategies. The report highlights instances where the program demonstrates effectiveness and explores potential enhancements. Additionally, it compares shared and distributed memory models in parallel programming.

## Problem statement

The challenge involves transitioning from a sequential search, as employed in Reversi for exploring potential moves one by one, to a parallel search. The objective is to enhance the efficiency of searching for the optimal current move and future moves by leveraging parallel processing. Helping accelerate the exploration of possible moves concurrently based on total number of CPU cores in the system.

## Logic and workings of Reversi

Reversi is an 8x8 grid board game where players use white and black counters to claim territory. Legal moves involve "hopping" over an opponent's counter vertically, horizontally, or diagonally, landing in an empty space. Placing a counter on the opposite side of an opponent's counter flips all intervening counters to the player's color. Positions on the board have varying score values, influencing desirability.

## How well my program works

The implemented program has successfully transformed the initially provided sequential search algorithm into a parallel search. Leveraging the full processing capacity of the system's available processor cores to process each move efficiently.

In the designed system, the mailbox is configured with default values of 0 for both input and output buffers. Additionally, semaphores are initialised with values of 0 for "item available" and 1 for "mutex." The overall semaphore value for available space is a constant set at 100, establishing a connection with the provided previous mailbox configuration. Two essential functions are employed within the mailbox to facilitate the exchange of data between parent and child processes.

The sending functionality incorporates a critical section enclosed by semaphores. It initiates by waiting for the availability of the space semaphore and subsequently waits for the mutex semaphore. Access to the critical section is granted only when both semaphores signal availability, enabling the addition of items to the buffer using predefined variables. The in index is then incremented, ensuring it remains within the bounds of the maximum mailbox size, thereby marking the entry point for subsequent additions to the bounded buffer. Following the configuration of values within the mailbox, the critical section concludes, signaling the availability of both the mutex and item available semaphores. Conversely, the receiving functionality also encompasses a critical section enclosed by semaphores. The "item available" and "mutex" semaphores wait for access signals before entering the critical section, where items are extracted from the buffer. The out index is similarly incremented, constrained within the limits defined for the in index. Following the manipulation of values within the mailbox, both the mutex and space available semaphores are signaled, indicating their renewed availability. The strategic use of semaphores serves to encapsulate critical sections of the program, preventing simultaneous edits to memory by multiple sources. This approach helps restrict and ensure safe access to the shared resources, enhancing the overall robustness and reliability of the program.

The parallel search begins by forking, enabling the parent to utilise a mailbox receiver to determine the best moves. Each child sends information to the parent via mailbox, initialises a move score, and executes a loop for available moves. After waiting for a processor signal, it forks again for parallelism and uses the alphaBeta function to calculate move scores. The scores are sent via the mailbox with a semaphore indicating processor availability. In the parent section, a screen print precedes waiting for mailbox input, followed by another print upon receipt. The total number of moves is updated, and move scores are compared and replaced if superior. This process is iterated for the number of moves to select the optimal move, updating the move pointer accordingly.

Code can be seen in Appendices 1 along with line-by-line commentary.

## Analysis and future improvements

My initial thoughts to test the performance of parallelism within my program were going to be achieved by limiting the virtual machine (VM) to one processor core and running my program. Doing this would mean the program is forced to run from the single core. I would time the programs execution of calculating ply 0 three times to get an average and then compare against the average time taken to run the program using four cores. The problem with using that method is the single core would also be responsible for hosting the VMs operating system (OS). This means the data could be skewed, as the single core will be handling multiple processes at a time, whereas when we use multiple cores, those processes will be split among all available cores.

Instead, I decided to look through the code for how it determines how many cores to use. We use the function multiprocessor\_wait to see when the semaphore processorAvailable denotes that a processor is available in the parallelSearch function see figure 1. Going to the definition for semaphore processorAvailable shows us figure 2, from this we can ascertain that processorAvailable is initialised by a function returning an integer value passed into the initSem function. Going to the definition of multiprocessor\_maxProcessors see figure 3, we can see that a system function is returned to get the number of processors and this system function is also of type int, this enables me to just return 1 and return 4 respectively from this function as seen in figure 4 to get the same result however only the program will be limited and not the entire VM. This should help eliminate the data skew from running the VM on one core where that single core is responsible for the OS as well as execution of the search algorithm.

The results for this test will be presented below.

The single processor presented results as an average of 400,000 positions processed per second on my machine. Compare this to my quad processor results which gave us an average of 1.5 million positions processed per second. This is a massive jump from the original 400,000 given for one core performance. These results were calculated by running the program multiple times and dividing the number of positions searched by the time taken adding the values together and dividing by the number of times the program ran. While these results do lack a bit of accuracy as not using tenths of seconds it did allow me to get a baseline average across multiple executions.

Forking of processes is a resource-intensive task compared to creating threads, which have lower creation costs and more efficient context switching, making them better suited for system performance. Threads give faster and smoother implementation compared to forking and threads also offer the advantage of shared memory. Helping address access control issues currently present when forking processes. Unlike forking processes, threads lack hierarchy or dependency concerns, meaning a fault in an earlier spawned thread doesn't necessarily impact subsequent threads. Nonetheless, threads carry the potential risk of corruption due to unrestricted memory accessibility. Another future improvement for the program would be creation of a local multiplayer mode where people could play against each other at the same machine. This would make for a fun experience and gives the program another level of desirability by offering more modes.

## Compare shared memory and distributed memory models in parallel programming

Parallel programming is crucial for optimising the computational capabilities of modern systems. The shared memory model involves processors accessing a unified global address space, facilitating direct interaction with shared variables. Communication through shared variables simplifies programming, particularly with frameworks like OpenMP, and synchronisation mechanisms, such as locks and barriers, are simpler to implement. However, scalability becomes a concern with an increasing number of processors, leading to contention for shared resources.

In contrast, the distributed memory model assigns each processor its local memory, requiring explicit message passing for communication. Although more complex, this approach offers better scalability as each processor operates independently, reducing contention. The programming model, exemplified by libraries like MPI, involves explicit management of data distribution and synchronisation, adding complexity. Hybrid models, combining elements from both paradigms, are employed in specific environments to leverage the strengths of each approach. The choice between shared and distributed memory models depends on factors like system architecture, processor count, and the nature of the parallel algorithm, emphasising the need for a nuanced understanding of their merits and drawbacks in parallel programming.

## Human Factors involved within the use of the system

There are no human factors involved within the use of my system as it is a parallel implementation of a game. The only time another human would be involved is if I made the game local multiplayer so the user could play against a person on the same machine, but it currently doesn’t have this functionality.

## Appendices:

## Appendix 1 - Code

Full code is available below.

[GitHub Repository Link](https://github.com/CM30019559/reversi_parallel)

Direct URL: <https://github.com/CM30019559/reversi_parallel>

Line by line changes.

### Mailbox.h

#define MAX\_MAILBOX\_DATA 100 // Constant variable to set number of total entries allowed to be stored in mailbox\_data to 100

typedef struct triple\_t {

int result;

int move\_no;

int positions\_explored;

} triple;

typedef struct mailbox\_t {

triple data[MAX\_MAILBOX\_DATA]; // Array of elements set to size of MAX\_MAILBOX\_DATA (100 currently), each element is a triple used to store result, move\_no and positions\_explored

int in; // Implements input index for bounded buffer, putting data in at this slot

int out; // Implements output index for bounded buffer, sending data out from this slot

sem\_t\* item\_available; /\* is there data in the mailbox? \*/

sem\_t\* space\_available; /\* is there space for more data in the mailbox? \*/

sem\_t\* mutex; /\* access to the mailbox. \*/

struct mailbox\_t\* prev; /\* pointer to previous mailbox. \*/

} mailbox;

### Mailbox.c

static mailbox\* mailbox\_config(mailbox\* mbox, mailbox\* prev)

{

mbox->in = 0; // Initialise in index to 0

mbox->out = 0; // Initialise out index to 0

mbox->prev = prev; // Set current pointer of prev to new pointer for prev

mbox->item\_available = multiprocessor\_initSem(0); // Initialise semaphore for item\_available using function multiprocessor\_initSem to initialise to a value of 0

mbox->space\_available = multiprocessor\_initSem(MAX\_MAILBOX\_DATA); // Initialise semaphore for space\_available using function multiprocessor\_initSem to initialise to a value of MAX\_MAILBOX\_DATA

mbox->mutex = multiprocessor\_initSem(1); // Initialise semaphore for mutex using function multiprocessor\_initSem to initialise to a value of 1

return mbox;

}

void mailbox\_send(mailbox\* mbox, int result, int move\_no, int positions\_explored) // mailbox\_send is a producer to the shared buffer, sending the result, move\_no and positions\_explored to the shared buffer

{

// My code

multiprocessor\_wait(mbox->space\_available); // Uses function multiprocessor\_wait to wait for semaphore space\_available to be available stopping us from overflowing the mailbox

multiprocessor\_wait(mbox->mutex); // Uses function multiprocessor\_wait to wait for semaphore mutex to be available to allow access to the mailbox

mbox->data[mbox->in].result = result; // Adding the data through the in index to the shared buffer for the result value

mbox->data[mbox->in].move\_no = move\_no; // Adding the data through the in index to the shared buffer for the move\_no value

mbox->data[mbox->in].positions\_explored = positions\_explored; // Adding the data through the in index to the shared buffer for the positions\_explored value

mbox->in = (mbox->in + 1) % MAX\_MAILBOX\_DATA; // This is a loop to increment in index by 1, divide by MAX\_MAILBOX\_DATA and place remainder into in index ensuring we don't overflow

multiprocessor\_signal(mbox->mutex); // Uses function multiprocessor\_signal to signal that an item is being added to the buffer by the semaphore mutex and mutex is now available

multiprocessor\_signal(mbox->item\_available); // Uses function multiprocessor\_signal to signal that an item is now available in the buffer using the semaphore item\_available

}

void mailbox\_rec(mailbox\* mbox, int\* result, int\* move\_no, int\* positions\_explored) // mailbox\_rec is a consumer from the shared buffer, retrieving the result, move\_no and positions\_explored from the shared buffer

{

// My code

multiprocessor\_wait(mbox->item\_available); // Uses function multiprocessor\_wait to wait for semaphore item\_available to say there is an item available

multiprocessor\_wait(mbox->mutex); // Uses function multiprocessor\_wait to wait for semaphore mutex to be available

\*result = mbox->data[mbox->out].result; // Removing the data through the out index from the shared buffer for the result value

\*move\_no = mbox->data[mbox->out].move\_no; // Removing the data through the out index from the shared buffer for the move\_no value

\*positions\_explored = mbox->data[mbox->out].positions\_explored; // Removing the data through the out index from the shared buffer for the positions\_explored value

mbox->out = (mbox->out + 1) % MAX\_MAILBOX\_DATA; // This is a loop to increment out index by 1, divide by MAX\_MAILBOX\_DATA and place remainder into out index ensuring we don't overflow

multiprocessor\_signal(mbox->mutex); // Uses function multiprocessor\_signal to signal that an item has been added to the buffer by the semaphore mutex and mutex is now available

multiprocessor\_signal(mbox->space\_available); // Uses function multiprocessor\_signal to signal that space is now available in the buffer using the semaphore space\_available

}

### Paro64bit.c

#if !defined(SEQUENTIAL)

int parallelSearch(int\* totalExplored, int\* move,

int best, int\* l, int noOfMoves,

BITSET64 c, BITSET64 u, int noPlies, int o, int minscore, int maxscore)

{

// My code

int pid = fork(); // Parent process forks to create a child to process each move on available cores

if (pid == 0) // If statement checks whether pid is a child as children processes have pid of 0

{

// Child is the source which spawns each move on a separate core

int i, currentMove; // Initialise variables to store values for the iterator in the loop responsible for looping noOfMoves value and to store currentMove to send to parent

for (i = 0; i < noOfMoves; i++) // Use noOfMoves to iterate through total number of potential moves

{

multiprocessor\_wait(processorAvailable); // Uses function multiprocessor\_wait to wait for semaphore processorAvailable to be available denoting there is a processor core free

if (fork() == 0) // Linux always forks onto available processor, this will force each move to be processed on it's own core

{

// Children search for move using alphaBeta

currentMove = alphaBeta(l[i], c, u, noPlies, o, minscore, maxscore); // Using alphaBeta function work out the currentMove score, passing array of list of moves (l[i]), current board (c), board state after move (u), number of game tree plies, colour (o) and min/max score

mailbox\_send(barrier, currentMove, i, positionsExplored); // Passes currentMove, current index (i) and positionsExplored back via mailbox\_send using barrier to point to mailbox to use

multiprocessor\_signal(processorAvailable); // Signal that a processor is available, notifying a wait function that it may continue

exit(); // Exit child process, move has been sent to parent

}

}

exit(); // Exit child process, all potential moves processed

}

else

{

// Parent is the sink process which awaits any move to be returned and remembers best move score

int i, move\_score, move\_index, positionsExplored; // Initialise variables to store index in the loop, move\_score, move\_index and positionsExplored

for (i = 0; i < noOfMoves; i++) // Use noOfMoves to iterate through total number of moves sent to parent

{

printf("parent waiting for a result\n"); // Output message stating parent has received a result

mailbox\_rec(barrier, &move\_score, &move\_index, &positionsExplored); // Receives the move from mailbox and stores in variables created above to compare best move

printf("... parent has received a result: move %d has a score of %d after exploring %d positions\n", move\_index, move\_score, positionsExplored); // Output message stating the potential move score and number of explored positions

\*totalExplored += positionsExplored; // Increment total number of explored positions by positionsExplored this time

if (move\_score > best) // If statement checks if current move score is better than current best move score

{

best = move\_score; // Make the best value score equal to the current move score

\*move = l[move\_index]; // Set move pointer to index in list of moves current index

}

}

}

return best; // Return the best move, this function is called by decideMove so this returns the best move to decideMove function which will make the move

}

#endif

## Appendix 2 - Video Link

[Demonstration Video Link](https://southwales.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=80dc252f-cecf-4dd9-b6f7-b0c401844125)

Direct URL: https://southwales.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=80dc252f-cecf-4dd9-b6f7-b0c401844125

## Figures

### Figure 1



Showing function multiprocessor\_wait function being called to wait until processorAvailable semaphore is available denoting a processor is available to be tasked.

### Figure 2

A computer screen with white text

Description automatically generated

Showing definition of processorAvailable and initialization and proving that processorAvailable is initialised by passing an int value to initSem function.

### Figure 3

A screenshot of a computer

Description automatically generated

Showing that function called to check number of processors using a system function returns an int value.

### Figure 4

A screenshot of a computer program

Description automatically generated

Showing how I intend to control the max number of processors the system is told we have by only returning an int value instead of using system function to find out.