Parallel and Concurrent Programming

Isaac.J

Student number - 30005209

Contents

List of Figures

1 Overall System Produced

1.1 Task Overview

The task for the author is to convert the currently sequential version of the game Reversi into a parallel program. The original source code can be found at https://github.com/gaiusm/reversi

1.2 Further Details of Task

The first level of the game tree (ply 0) must be made parallel. All moves must be evaluated above ply 0 using the origin sequential algorithm. A mailbox library must also be implemented which will provide a simple message passing primitives. Parallelism is to be restricted to ply 0 only. The parallel solution must also be limited to the number of cores available on the host machine. The parallel solution should fork a producer process with another fork process for every move. This producer must also be limited to the number of cores found on the machine. Move results, when found, should be passed back to the original process.

1.3 System Produced

1.3.1 Mailbox

The mailbox itself has default variables of 0 for both in and out buffers. With semaphores being set to 0 (item available) and 1 (mutex). With the total semaphore value for space available being set to a constant value of 100. The previous is set to the supplied previous mailbox. (Figure: 1)

The send functionality of the mailbox has a critical section surrounded by semaphores. It begins by waiting available space via semaphore. Then waiting for the mutex to trigger. Once both of these allow access to the critical section is enabled. This part allows items to be added to the buffer via supplied variables. The in variable is then incremented (constrained by it's max mailbox size). This denotes the entry-point for future additions to the buffer. It must be constrained by the max size as the buffer is circular. Once that has been set the critical section has ended and signals are allowed to be sent to the mutex and item available semaphores respectively.(Figure: 4) + The receive functionality of the mailbox also contains a critical section surrounded by semaphores. Item available and mutex must wait until access can be granted (via signals). The critical section removes items from the buffer then the placement for the out position is incremented and bound by max size similar to the in buffer previously explained. This concludes the critical section. Allowing signals to be sent to both mutex and space available. (Figure: 5)

The semaphores surround the critical sections to avoid contamination of code where code could be edited by multiple sources simultaneously. These are in place to make sure that access is restricted in a safe manner

1.3.2 Parallel Search

The search algorithm begins by forking. This is so the parent is able to utilise the mailbox receiver to work out the best of the found moves. With children being responsible for sending the information via mailbox send to aforementioned parent.

The child, instantiating but not initialising move score beings a for loop for the number of available moves. Incrementing both i (which will denote the move number) and the listOfMoves pointer.Allowing them to point at the next possible move or increment the current move number respectively. It then will wait for an available processor to signal it is available for use.(Figure: 8).

The child will then fork again for each iteration in the for loop (to allow parallelism). Each child will then use the alphaBeta function, supplied with appropriate variables, to work out the current score of a potential move. This move is then sent via the mailbox. A semaphore signal is then send to denote a processor is available. The child is then closed. With the final child (at this level) closing again afterwards now all potentials have been passed.

Within the parent section of the code a print to screen is performed prior to waiting to receive input from the mailbox (sent within the children). Once received another print to screen is performed and the total number of moves added to the running total. The move score is compared (and replaces the current best should it be better ranking). This process within the parent is then repeated for the number of moves so that the best move can be chosen. When selected the move with the highest move score is set as the move, via pointer, by making the move equal to the listOfMoves(pointer) incremented by the move index passed via the mailbox.

1.4 Resulting Source Code

For full code please visit https://github.com/30005209/reversi

1.4.1 mailbox.c

Please refer to figures: 1), 2), 3), 4), 5) for changed code.

1.4.2 mailbox.h

Please refer to figures: 6), 7) for source code

1.4.3 paro64bit.c

Please refer to figures: 8), 9), 10) for source code

2 Analysis and Future Improvements

2.1 Analysis

2.2 One Core Comments

Whilst one core is being used the program is effectively running sequentially. Due to the semaphores blocking access and the available processors only having a maximum value of one (when one core is used) there cannot be two active. This serves as a great baseline for comparisons. This is because the code will be identical and the only change will be a shift from sequential to a parallel one with more cores allowing even more parallelism.

2.3 Two Core Comments

Compared to the one core test there was a dramatic increase in the number of moves searched. An increase of 30, 441, 643 moves. The pattern was attempted to be followed, however, as more cores were used different moves were selected by the program. This can also been seen as the printed output of all non-single-core trials had output that was non uniform regarding the order of move number prints. This showed that certain move sets completed their searches sooner than others (highlighting the parallelism). There were many times the output of the moves was done within one second. The author noted the change in the highest positions searched with this one second limit. Comparing the two core system saw an increase of 68, 654 more searches within that time restriction. Another feature noted was the longest time spent waiting for the full feedback from the program. Originally having this stall time of 24 seconds the two core system brought that number down to 15 seconds. With the overall time for the entire running of the program being comparable within a second.

2.4 Four Core Comments

The four core system had a higher total wait time compared to all other systems of 110. It however also had the highest time waiting for results to come back from the program with a high time of 38 seconds, 14 seconds slower than the original single core / sequential system. There however, was two dramatic increases compared to both previous systems. The maximum positions searched within 1 second dramatically increased by 568, 905 to 853, 222 when compared the single core system. This dramatic uptick in speed is also clearly shown with the total moves searched which reached a total of 191, 984, 180 - 103, 400, 045 more position searched than the last two systems combined.

2.5 Six Core Comments

There was a decrease in the total searched positions when compared to the four core system. The author this believes that this was due to the program making better moves overall and being the only trail which beat the author. The other rationale for why the lower total searched positions is not seen as a negative is both the total time searching for positions and the highest time waiting for a full response was notably lower. There also was a impressive increase for the maximum positions searched under one second beating the original core system by over a factor of 10. Reaching a high of 3, 311, 172 opposed to 284,317 respectively.

2.6 Overall Comments

The increase of cores was very clearly a positive factor for the code-base as a whole, showing both the system running well in parallel as well as the tangible benefits of such a system. Beating the author as well as being able to do it at a faster rate with slower wait times for any other potential players. The Four-Core system time increase is believed to be down to a longer overall game with a larger total number of moves played. Further testing could highlight this. If possible the same moves being taken could show the upticks in speed. The long time seems to be showing a challenge posed to the author that was not as pronounced as on lower cored systems. Though across this data set there is a notable difference between sessions one should take note of Amdahl's Law where states how parallelism can only produce an increase of speeds in multi-cored/multi-threaded systems with code that is suited for parallelism. The execution time irrespective of total forked processes or threads made will reach a maximum speed (minimum time) as there will always be sections of code that is not able to be / not suited for parallelism .

Please refer to 11) for full details

2.7 Future Improvements

Forking processes is an intense process especially when compared to threads. Their lighter creation costs combined with their light context switching abilities make it better suited as a system than forking processes. It would be likely to have a much faster and smoother implementation. Threads also have the additional benefit of all memory being shared memory which helps alleviate problems with granting or restricting access via forks this would. With all threads there is also no hierarchy or dependency (inter-dependency or otherwise) which means should a fault occur on a thread spawned earlier it will not necessarily cause problems for later threads. Where as a parent thread error could close the entire code-base down due to the fact that should a parent process end

so too do all it's children. however, come with the potential issue of corruption. With all memory being accessible able this could, and is more likely to, result in memory being corrupted due to it all being accessible.

3 Line by Line Commentary and Testing

```c
#define mailbox_c

#include "mailbox.h"

#define NO_MAILBOXES 30

static void *shared_memory = NULL;
static mailbox *freelist = NULL;  /* list of free mailboxes.  */


/*
 *  initialise the data structures of mailbox.  Assign prev to the
 *  mailbox prev field.
 */

static mailbox *mailbox_config (mailbox *mbox, mailbox *prev)
{
  // Set the in and out to 0
  mbox->in = 0;
  mbox->out = 0;

  // Set the current pointer of previous to supplied pointer for previous
  mbox->prev = prev;

  // Initialise the semaphore for item available
  mbox->item_available = multiprocessor_initSem(0);

  // Initialise the semaphore for space available using MAX_MAILBOX_DATA denoting the total max size
  mbox->space_available = multiprocessor_initSem (MAX_MAILBOX_DATA);

  // Initialise the mutex semaphore
  mbox->mutex = multiprocessor_initSem (1);

  return mbox;
}
```

Figure 1: mailbox.c 0

```
/*
 *  init_memory - initialise the shared memory region once.
 *                It also initialises all mailboxes.
 */

static void init_memory (void)
{
  if (shared_memory == NULL)
    {
      mailbox *mbox;
      mailbox *prev = NULL;
      int i;
      _M2_multiprocessor_init ();
      shared_memory = multiprocessor_initSharedMemory
        (NO_MAILBOXES * sizeof (mailbox));
      mbox = shared_memory;
      for (i = 0; i < NO_MAILBOXES; i++)
        prev = mailbox_config (&mbox[i], prev);
      freelist = prev;
    }
}


/*
 *  init - create a single mailbox which can contain a single triple.
 */

mailbox *mailbox_init (void)
{
  mailbox *mbox;

  init_memory ();
  if (freelist == NULL)
    {
      printf ("exhausted mailboxes\n");
      exit (1);
    }
  mbox = freelist;
  freelist = freelist->prev;
  return mbox;
}
```

Figure 2: mailbox.c 1

```
/*
 *  kill - return the mailbox to the freelist.  No process must use this
 *         mailbox.
 */

mailbox *mailbox_kill (mailbox *mbox)
{
  mbox->prev = freelist;
  freelist = mbox;
  return NULL;
}


/*
 *  send - send (result, move_no, positions_explored) to the mailbox mbox.
 */
```

Figure 3: mailbox.c 2

```
/*
 *  send - send (result, move_no, positions_explored) to the mailbox mbox.
 */

void mailbox_send (mailbox *mbox, int result, int move_no, int positions_explored)
{

  // Wait until the space semaphore denotes it is available
  multiprocessor_wait(mbox->space_available);

  // Wait until the mutex semaphore denotes it is available
  multiprocessor_wait(mbox->mutex);

  // Critical Section Begin

  // Add Item to buffer
  mbox->data[mbox->in].result = result;
  mbox->data[mbox->in].move_no = move_no;
  mbox->data[mbox->in].positions_explored = positions_explored;

  // Increment in, divide by max, place remainder into mbox in
  mbox->in = (mbox->in +1) % MAX_MAILBOX_DATA;

  // Critical Section End

  multiprocessor_signal(mbox->mutex);
  multiprocessor_signal(mbox->item_available);
}
```

Figure 4: mailbox.c 3

```
/*
 *  rec - receive (result, move_no, positions_explored) from the
 *        mailbox mbox.
 */

void mailbox_rec (mailbox *mbox,
                  int *result, int *move_no, int *positions_explored)
{

  // Wait for semaphore to denote item is available
  multiprocessor_wait(mbox->item_available);

  // Wait for the semaphore to denote mutex is allowing access
  multiprocessor_wait(mbox->mutex);

  // Critical Section begin

  // Remove item from buffer

  *result = mbox->data[mbox->out].result;
  *move_no = mbox->data[mbox->out].move_no;
  *positions_explored = mbox->data[mbox->out].positions_explored;

  // Increment in, divide by max, place remainder into mbox in
  mbox->out = (mbox->out +1) % MAX_MAILBOX_DATA;

  // Critical Section end
  multiprocessor_signal(mbox->mutex);
  multiprocessor_signal(mbox->space_available);
}
```

Figure 5: mailbox.c 4

```c
/*  mailbox.h provides a very simple mailbox datatype.
 *  Gaius Mulley <gaius.southwales@gmail.com>.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/sysinfo.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>

#include "multiprocessor.h"

// Set the max size for mailbox data as a constant
#define MAX_MAILBOX_DATA 100

typedef struct triple_t {
  int result;
  int move_no;
  int positions_explored;
} triple;


typedef struct mailbox_t {
  triple data[MAX_MAILBOX_DATA];
  int in, out;
  sem_t *item_available;    /* are there data in the mailbox?  */
  sem_t *space_available;   /* space for more data in the mailbox.  */
  sem_t *mutex;             /* access to the mailbox.  */
  struct mailbox_t *prev;  /* previous mailbox.  */
} mailbox;
```

Figure 6: mailbox.h 0

```
#if !defined(mailbox_h)
#  define mailbox_h
#  if defined(mailbox_c)
#     if defined(__GNUG__)
#        define EXTERN extern "C"
#     else /* !__GNUG__.  */
#        define EXTERN
#     endif /* !__GNUG__.  */
#  else /* !mailbox_c.  */
#     if defined(__GNUG__)
#        define EXTERN extern "C"
#     else /* !__GNUG__.  */
#        define EXTERN extern
#     endif /* !__GNUG__.  */
#  endif /* !mailbox_c.  */


/*
 *  init - create a single mailbox which can contain a single triple.
 */

EXTERN mailbox *mailbox_init (void);


/*
 *  send - send (result, move_no, positions_explored) to the mailbox mbox.
 */

EXTERN void mailbox_send (mailbox *mbox,
                          int result, int move_no, int positions_explored);


/*
 *  rec - receive (result, move_no, positions_explored) from the
 *        mailbox mbox.
 */

EXTERN void mailbox_rec (mailbox *mbox,
                         int *result, int *move_no,
                         int *positions_explored);

#  undef EXTERN
#endif /* !mailbox_h.  */
```

Figure 7: mailbox.h 1

```
#if !defined(SEQUENTIAL)
int parallelSearch (int *totalExplored, int *move,
                    int best, int *listOfMoves, int noOfMoves,
                    BITSET64 c, BITSET64 u, int noPlies, int colour, int minscore, int maxscore)
{

  // Parent forks
  int pid = fork();

  // Check if child (as children have a pid of 0)
  if (pid==0)
    {
      // Children of the original parent will proceed to spawn moves (move_score)  on cores
      int move_score;

      // Iterate, for the size of the number of potential moves, both the current index
      // and the current move being pointed to via the listOfMoves pointer
      for (int i = 0; i < noOfMoves; i++, *(listOfMoves)++)
        {

          // Wait for available processor via a multiprocessor semaphore
          multiprocessor_wait(processorAvailable);
```

Figure 8: paro64bit.c 0

```
        // As linux always forks on to an availble processor, this will be handled via kernel
        if (fork() == 0)
          {
            // Children search for move (move_score)

            // Using alphaBeta function work out move_score of the currently passed list of moves pointer
            // referencing the current board (c) and its state after the move (u) for the current colour
            // bounded via the minscore and maxscore variables also passed to the parallel search function
            move_score = alphaBeta(*listOfMoves, c, u, noPlies, colour, minscore, maxscore);

            // Pass move_score, i +1 (an increase of 1 has been added as to count from 1 rather than 0),
            // and positions_expored back via mailbox send
            // A pointer for the mailbox in question is denoted by the barrier pointer
            mailbox_send(barrier, move_score, i+1, positionsExplored);

            // Signal processorAvailable semaphore this will notify that a wait semaphore for the processor
            // may continue
            multiprocessor_signal(processorAvailable);

            // Close child process as the move has been worked out, sent and the semaphore singnaled now the
            // critical section has passed
            exit(0);
          }
      }

    // Close child process once all move potentials have been passed
    exit(0);
```

Figure 9: paro64bit.c 1

```
      // Close child process once all move potentials have been passed
      exit(0);
    }
  else
    {

      // parent is in the sink which waits for any move to be returned and remembers the best move score

      int i, move_score, move_index, positions_explored;

      // Similarly to the child above iterate over all possible moves
      for (i=0; i < noOfMoves; i++)
        {
          // Print that the parent is currently waiting for a result
          printf("parent's waiting for result\n");

          // Once the mailbox has recieved input via a mailbox send (send from children) place the information gathered into the
          // local variables defined above
          mailbox_rec(barrier, &move_score,  &move_index, &positions_explored);

          // Print that a result has been recieved and denote both its score and total number of explored positions as well as which move set it is
          printf("...parent has recieved a result: move %d has a score of %d after exploring %d positions\n",
                 move_index, move_score, positions_explored);

          // Increment the total number of explored positions by the positions explored this search
          *totalExplored += positions_explored; // add count to the running total

          // Should the move score be better than the current best
          if (move_score > best)
            {
              // Make the best value = to the current move score
              best = move_score;

              // Set the pointer = to the index in listOfMoves - as if it's an array
              *move = listOfMoves[move_index];
            }
        }
    }

  return best;
}
#endif
```

Figure 10: paro64bit.c 2

	Total Moves Searched	Total Time	Max Positions Searched Sub 1 Second	Highest Time
One-Core	29071246	73	284317	24
Two-Core	59512889	72	352971	15
Four-Core	191984180	110	853222	38
Six-Core	118311261	63	3311172	17

Figure 11: Testing Results