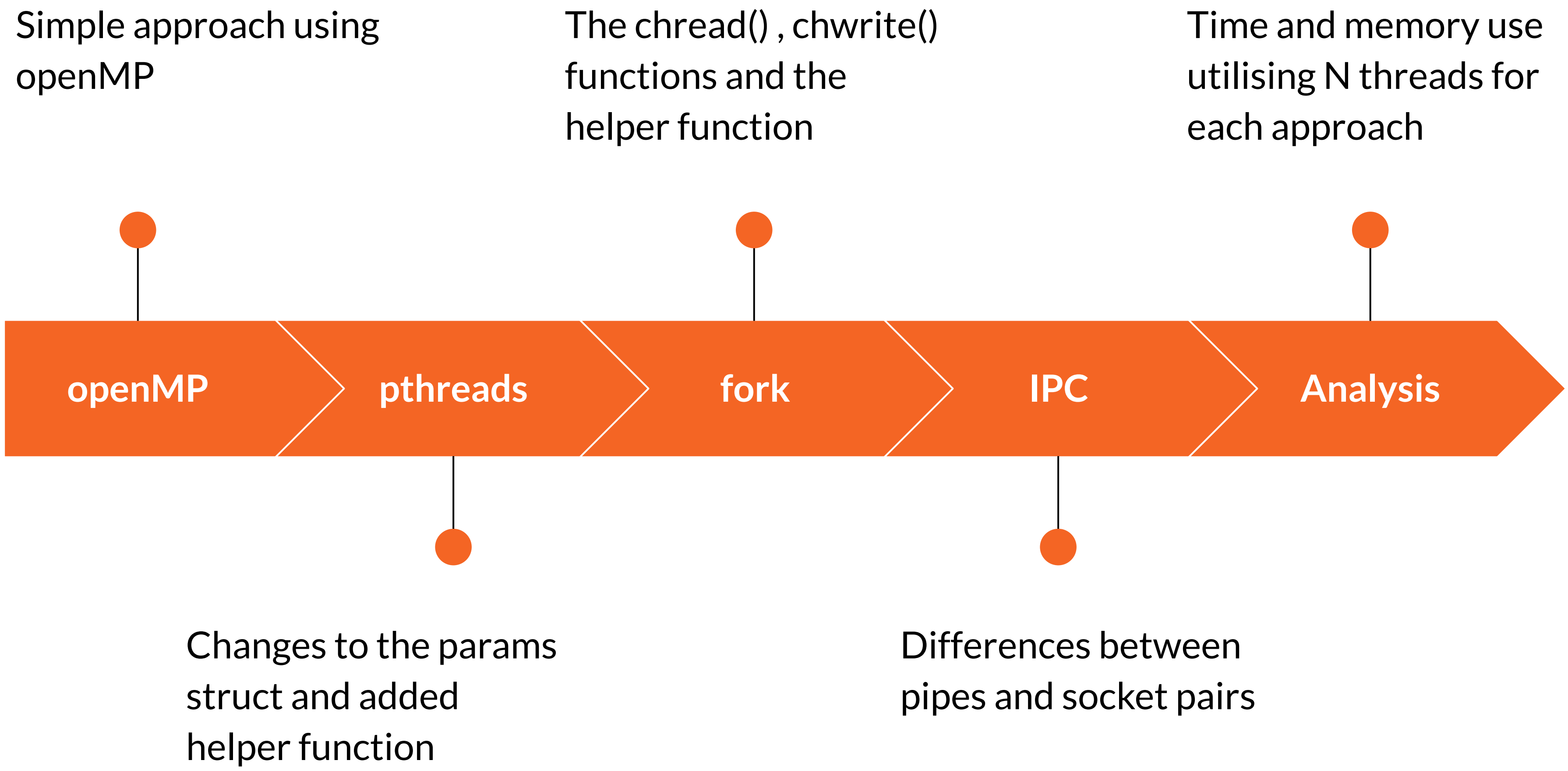
# 3325ENG: Parallelising Mandelbrot Set

Timothy Tadj s5178358
River Shanahan s5178094

# Slowest Function (Serial)

```
Initialisation time: 0.013456 seconds
Default computation time: 28.429960 seconds
Histogram colouring time: 0.585731 seconds
Writing to file time: 1.006984 seconds
```

- Computing the mandelbrot is the slowest operation
- We choose to parallelize this function to minimise run time


Simple approach using openMP
The chread() , chwrite() functions and the helper function
Time and memory use utilising N threads for each approach
openMP
pthreads
fork
IPC
Analysis
Changes to the params struct and added helper function
Differences between pipes and socket pairs

# Design

# Design - openMP

```
#include <omp.h>
↓↓↓↓↓
void mandelCompute(Parameters *p)
{
    double absz;
    double complex c, z;
    int i, j, k;
    omp_set_num_threads(p->numThreads);
    #pragma omp parallel for private(i, j, k, c, z, absz) shared(p)
    for (i = 0; i < p->height; i++) {
```

## Simplest multithreading implementation

- Include openMP
- Add preprocessor in front of the first for loop

# Design - pthreads

```
#include <pthread.h>
↓↓↓↓↓
#define MAX_THREADS 64
typedef struct {
↓↓↓↓↓
    // added for pthreads
    int numThreads;
    pthread_t threads[MAX_THREADS];
} Parameters;
```

## Changes to Parameters struct

- Added *numThreads* parameter that stores the number of threads we are using
- Added *thread* id array for threads to keep track of work they are responsible for

# Design - pthreads

```
int main(int argc, char *argv[]){
↓↓↓↓
    p.numThreads = numThreads;
↓↓↓↓
void mandelcompute_pthread(Parameters *p)
{
    for (int i = 0; i < p->numThreads; i++) {
     pthread_create(&p->threads[i], NULL,
            mandelComputeThread, (void *)p);
    }
    for (int i = 0; i < p->numThreads; i++) {
        pthread_join(p->threads[i], NULL);
    }
}
```

## Added function for pthreads

- Creates the thread to calculate the mandelbrot set
- Stores thread id in shared prams struct

# Design - pthreads

```
void* mandelComputeThread(void *arg){
    Parameters *p = (Parameters *)arg;
    ↓↓↓↓↓
    pthread_t thread_id = pthread_self();
    int thread_idx = 0;
    for (int i = 0; i < p->numThreads; i++) {
        if (p->threads[i] == thread_id) {
            thread_idx = i;
            break;
        }
    }
    int block_size = p->height / p->numThreads;
    int i_start = block_size * thread_idx ;
    int i_end = i_start + block_size;
    if (thread_id == p->threads[p->numThreads - 1]) {
        i_end = p->height;
    }
      for (i = i_start; i < i_end; i++) {
```

## Changed mandelCompute function

- Calculates the start point and block size based on the position of its thread id in the threads id array
- Account for case where last threads block size is uneven
- Go from *i_start* to *i_end* instead of 0 to *p->height*

# Design - fork

```
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
↓↓↓↓↓
#define MAX_PROCESSES 64
#define IPC_DIM 2
enum {READ, WRITE};
enum {CHILD, PARENT};
typedef struct {
↓↓↓↓↓
    // added for fork
    int numProcesses;
    int process_idx;
    char ipctype; // 'p' for pipe, 's' for socket
} Parameters;
```

## Changes to Parameters struct

- Added *numProcesses* parameter that stores the number of threads we are using
- Added *process_idx* to make program self
- Added *ipctype* to change interprocess communication method

# Design - fork

```
void mandelcompute_fork(Parameters *p){
    int n_processes = p->numProcesses;
    int n_children = n_processes-1;
    int fd[n_children][IPC_DIM];
    ↓↓↓↓↓
    //make pipes or sockets here
    ↓↓↓↓↓
    int process_idx = 0;
    p->process_idx = process_idx;
    int chunk_size = p->height / p->numProcesses;
    int i_start[p->numProcesses];
    int i_end[p->numProcesses];
    for (int i = 0; i < p->numProcesses; i++){
        i_start[i] = i * chunk_size;
        i_end[i] = (i + 1) * chunk_size;
    }
    i_end[p->numProcesses - 1] = p->height;
```

## Added function for fork

- Initialises the IPC
- Compute the *i_start* and *i_end* points for each process

# Design - fork

```
void mandelcompute_fork(Parameters *p){
    //ipc setup and chunk calculation
    ↓↓↓↓↓
    for (int i = 1; i < n_processes; i++) {
        child_id = fork();
        if (child_id == 0) {
            process_idx = i;
            break;
        }
    }
chunk_size = i_end[process_idx] -
            i_start[process_idx];
    //run compute and chread and chwrite
    //for children and parent
```

## Changed mandelCompute function

- Initialise child processes
- Keep track of the *process_idx*

# Design - fork

```
void mandelcompute_fork(Parameters *p){
↓↓↓↓↓ //child
    if (child_id == 0) {
        p->process_idx = process_idx;
        mandelComputeProcess(p);
        int* buf = &p->iterations[
              i_start[process_idx]*p->width];
        if (p->ipctype=='p'){
            chwrite(fd[process_idx-1][WRITE],
                     buf, chunk_size*p->width, 1024);
        }
        else if (p->ipctype=='s'){
            chwrite(fd[process_idx-1][CHILD],
                     buf, chunk_size*p->width, 1024);
        }
        exit(0);
    }
```

## Changed mandelCompute function

- Run the child process
- Compute mandelbrot set and than write to parent

# Design - fork

```
void mandelcompute_fork(Parameters *p){
↓↓↓↓↓ //parent
    else {
        mandelComputeProcess(p);
        for (int i = 0; i < n_children; i++) {
            chunk_size = i_end[i+1] - i_start[i+1];
            int* buf = &p->iterations[
                     i_start[i+1]*p->width];
            if(p->ipctype=='p'){
                chread(fd[i][READ],
                     buf, p->width*chunk_size, 1000);
            }
            else if(p->ipctype=='s'){
                chread(fd[i][PARENT],
                     buf, p->width*chunk_size, 1000);
        }}}
```

## Changed mandelCompute function

- Run the parent process
- Compute mandelbrot set and than read from child

# Design - chread/chwrite

```
int chread(int fd, int *buf, int total_ints, int chunk_size)
{
    int bytes_read = 0;
    for (int i = 0; i < total_ints ; i += chunk_size) {
       int ret = read(fd, &buf[i],
                    chunk_size*sizeof(int));
        bytes_read += ret;
        if (ret <0) {
            perror("read");
            return -1;
        }
    }
   ↓↓↓↓↓ \\ send remainder of message
    return bytes_read
}
```

## New chread and chwrite functions

- Fundamentally the same, just replace the *read()* with *write()*
- Read/write the specified chunk size
- chunk_size is the number of ints we send at a time

# Design - chread/chwrite

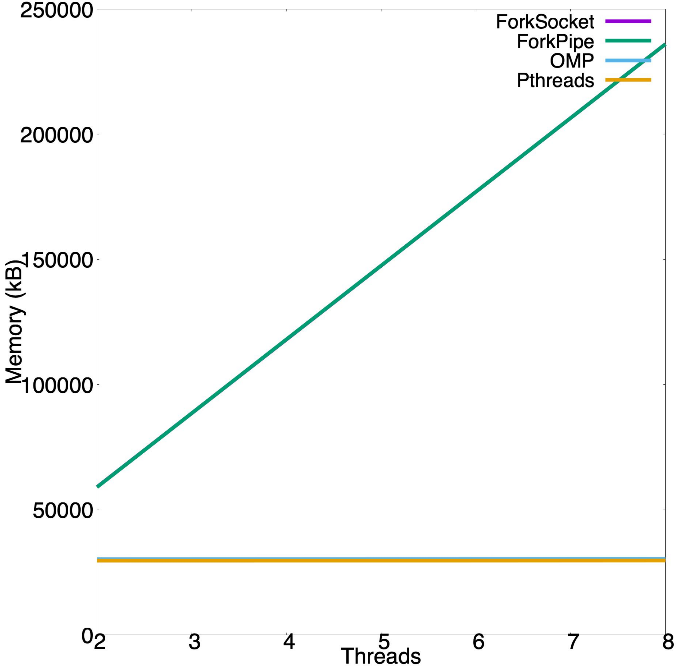
```
int chread(int fd, int *buf, int total_ints, int chunk_size)
{
↓↓↓↓↓
    if (bytes_read != total_ints*sizeof(int)) {
       int ret = read(fd, &buf[bytes_read-chunk_size-1],
                    total_ints*sizeof(int) - bytes_read);
        bytes_read += ret;
        if (ret <0) {
            perror("read");
            return -1;
        }
    }
    return bytes_read;
}
```

## New chread and chwrite functions

- Any numbers left we read/write
- We return bytes read/written or -1 if an error has occured

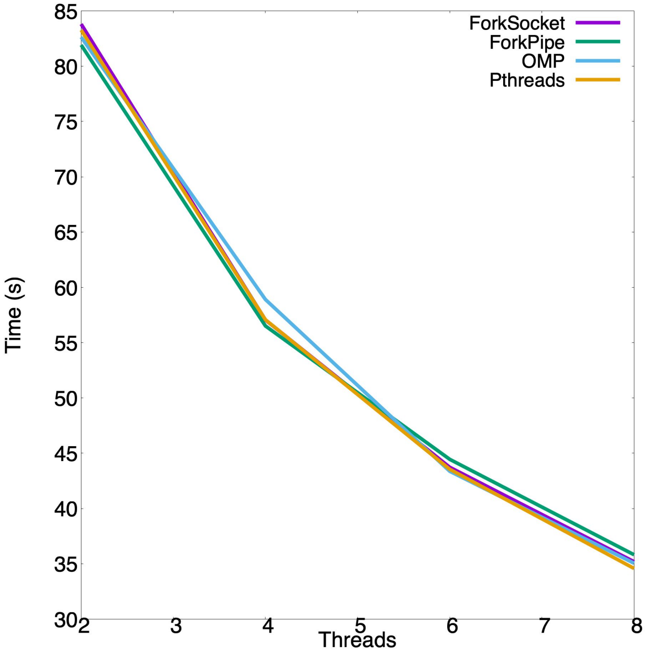
# Analysis

# Memory Analysis



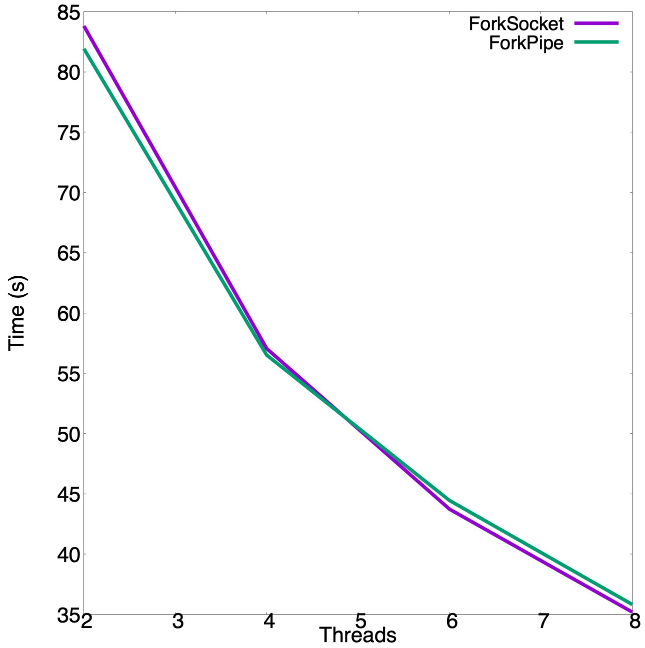

- Fork has a linear increase in memory usage (new processes)
- OMP and pthreads have a slight increase in memory usage

# Time Analysis



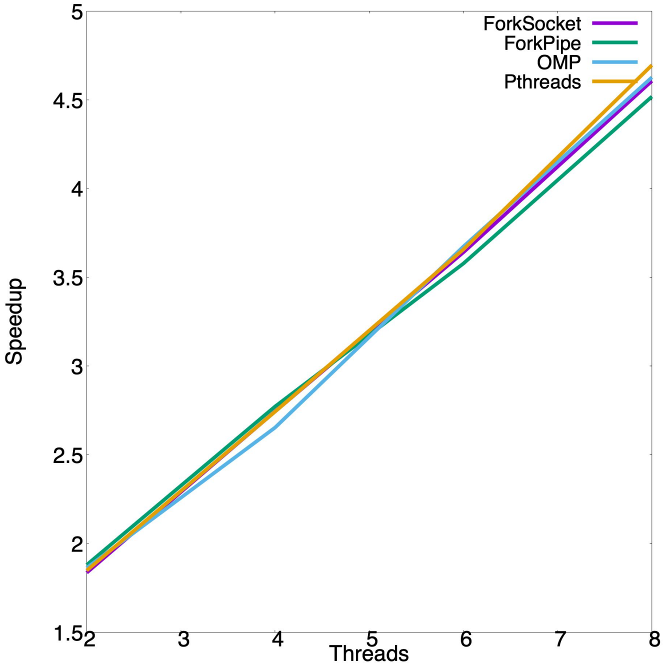

- Consistent ratio between different algorithms
- Time decreases as thread count increases
- A logarithmic decline means the speed will stagnate at a certain thread count

# Pipes vs Socketpair



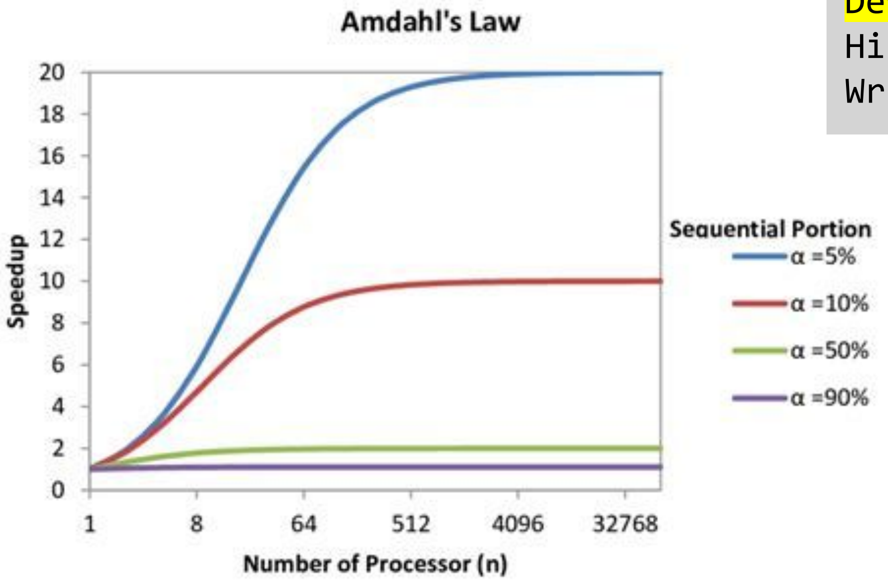

- Sockets are better when using more threads
- Pipes are better with less threads
- This could be due to a potentially longer initialisation time for pipes

# Amdahl's Law




- Relationship between speed achieved and base program
- Linear for up to 8 threads
- Will stagnate at a higher thread count

# Amdahl's Law




```
Initialisation time: 0.013456 seconds
Default computation time: 28.429960 seconds
Histogram colouring time: 0.585731 seconds
Writing to file time: 1.006984 seconds
```

- We have 5% sequential portion of run time.
- Could use up to 64 processes with linear gain

# Which method is most efficient?

pthreads

# Which method is Worst?

For MDSI fork is the worst option

# What will I use?

OpenMP!!!

We will now perform a live demonstration of a parallelisation of the Mandebrot set as well as the performance analysis