Concurrency with pthreads

David Hovemeyer

22 November 2019

Concurrency using processes

Processes created with fork can be used for concurrency, but processes are a heavyweight abstraction requiring significant resources:

They require:

Concurrency using processes

Processes created with fork can be used for concurrency, but processes are a heavyweight abstraction requiring significant resources:

They require:

- Address space data structures

Concurrency using processes

Processes created with fork can be used for concurrency, but processes are a heavyweight abstraction requiring significant resources:

They require:

- Address space data structures

- Open file table

Processes created with fork can be used for concurrency, but processes are a heavyweight abstraction requiring significant resources:

They require:

- Address space data structures

- Open file table

- Process context data

Concurrency using processes

Processes created with fork can be used for concurrency, but processes are a heavyweight abstraction requiring significant resources:

They require:

- Address space data structures

- Open file table

- Process context data

- Etc.

Processes created with fork can be used for concurrency, but processes are a heavyweight abstraction requiring significant resources:

They require:

- Address space data structures

- Open file table

- Process context data

- Etc.

Scheduling a process requires switching address spaces (possibly losing useful context built up in caches and TLB)

Threads

Threads are a mechanism for concurrency within a single
process/address space

A thread is a ''virtual CPU'' (program counter and registers):
each thread can be executing a different stream of instructions

Compared to processes, threads are lightweight, requiring only:

Threads

Threads are a mechanism for concurrency within a single process/address space

A thread is a ''virtual CPU'' (program counter and registers): each thread can be executing a different stream of instructions

Compared to processes, threads are lightweight, requiring only:

- Context (memory in which to save register values when thread is suspended)

Threads

Threads are a mechanism for concurrency within a single
process/address space

A thread is a ''virtual CPU'' (program counter and registers):
each thread can be executing a different stream of instructions

Compared to processes, threads are lightweight, requiring only:

- Context (memory in which to save register values when thread
  is suspended)

- A stack

Threads

Threads are a mechanism for concurrency within a single process/address space

A thread is a ''virtual CPU'' (program counter and registers): each thread can be executing a different stream of instructions

Compared to processes, threads are lightweight, requiring only:

- Context (memory in which to save register values when thread is suspended)

- A stack

- Thread-local storage (for per-thread variables)

Pthreads

Pthreads = ''POSIX threads''

Standard API for using threads on Unix-like systems

Allows:

- Creating threads and waiting for them to complete

- Synchronizing threads (more on this soon)

Can be used for both concurrency and parallelism (on multicore machines, threads can execute in parallel)

Some basic concepts:

pthread_t: the *thread id* data type, each running thread
has a distinct thread id

Thread attributes: runtime characteristics of a thread

- Many programs will just create threads using the *default*
  attributes

Attached vs. detached: a thread is *attached* if the program
will explicitly call pthread_join to wait for the thread
to finish.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Creates a new thread. Thread id is stored in variable pointed-to by *thread* parameter. The *attr* parameter specifies attributes (NULL for default attributes.)

The created thread executes the *start_routine* function, which is passed *arg* as its parameter.

Returns 0 if successful.

pthread_join

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Waits for specified thread to finish. Only attached threads can be waited for.

Value returned by exited thread is stored in the variable pointed-to by *retval*.

pthread_self

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Allows a thread to find out its own thread id.

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Changes the specified thread to be detached, so that its resources can be freed without another thread explicitly calling pthread_join.

Third version of the example web server: mt_webserver.zip on
course web page

Features:

Multithreaded web server

Third version of the example web server: mt_webserver.zip on
course web page

Features:

- Server will create a thread for each client connection

Multithreaded web server

Third version of the example web server: mt_webserver.zip on
course web page

Features:

- Server will create a thread for each client connection

- Created threads are *detached*: the server program doesn't
  wait for them to complete

Multithreaded web server

Third version of the example web server: mt_webserver.zip on
course web page

Features:

- Server will create a thread for each client connection

- Created threads are *detached*: the server program doesn't
  wait for them to complete

- No limit on number of threads that can be created

Multithreaded web server

Third version of the example web server: mt_webserver.zip on course web page

Features:

- Server will create a thread for each client connection

- Created threads are *detached*: the server program doesn't wait for them to complete

- No limit on number of threads that can be created

- Only the main function is different than previous versions

struct ConnInfo: represents a client connection:

```
struct ConnInfo {
  int clientfd;
  const char *webroot;
};
```

It's useful to pass an object containing data about the task the thread has been assigned to the thread's start function

The worker function (executed by client connection threads):

```
void *worker(void *arg) {
  struct ConnInfo *info = arg;

  pthread_detach(pthread_self());

  server_chat_with_client(info->clientfd, info->webroot);
  close(info->clientfd);
  free(info);

  return NULL;
}
```

A created thread detaches itself, handles the client request,
closes the client socket, frees its ConnInfo object,
then returns

Main loop:

```
while (1) {
  int clientfd = Accept(serverfd, NULL, NULL);
  if (clientfd < 0) {
    fatal("Error accepting client connection");
  }

  struct ConnInfo *info = malloc(sizeof(struct ConnInfo));
  info->clientfd = clientfd;
  info->webroot = webroot;

  pthread_t thr_id;
  if (pthread_create(&thr_id, NULL, worker, info) != 0) {
    fatal("pthread_create failed");
  }
}
```
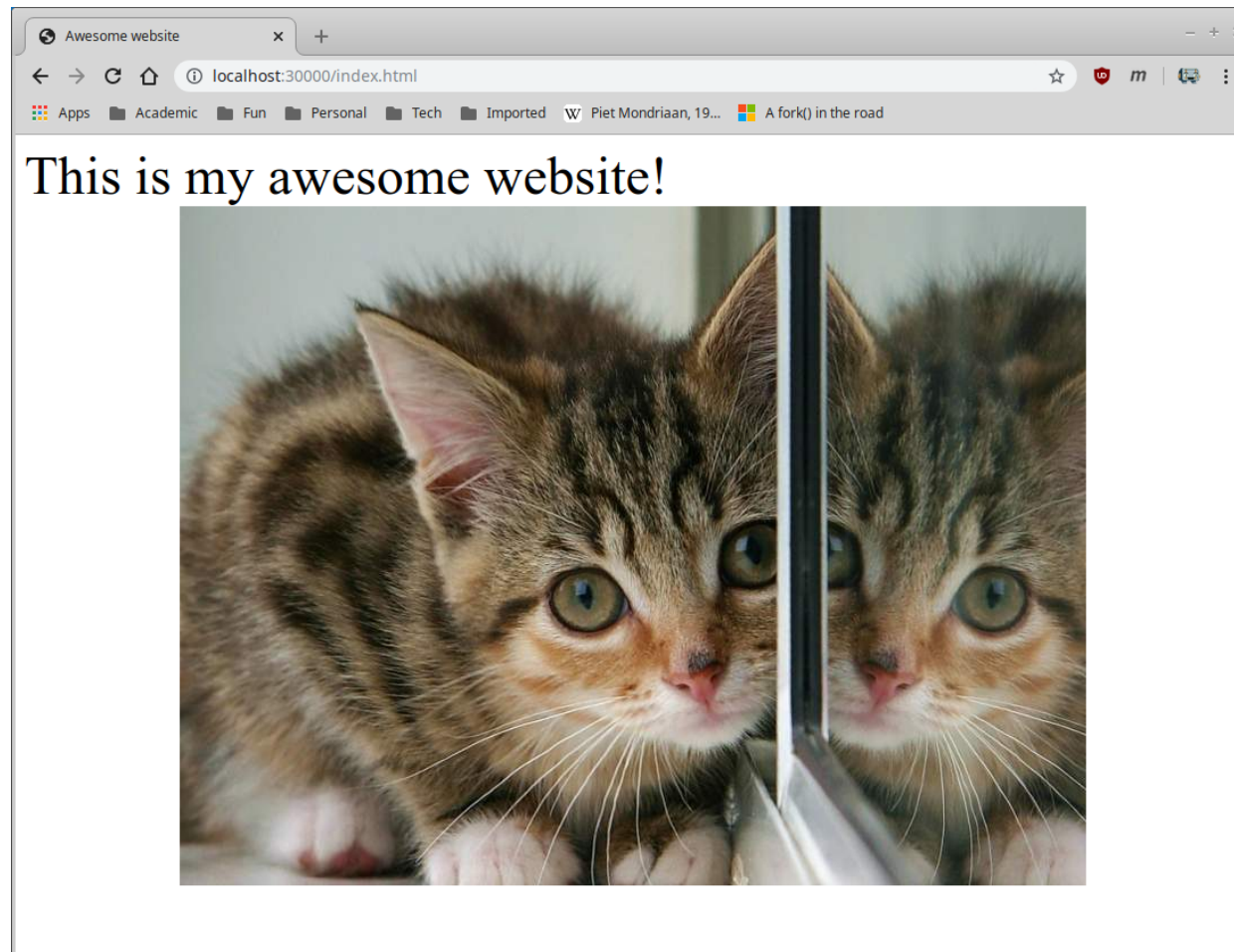
Compile and run the server:

```
$ gcc -o mt_webserver main.c webserver.c csapp.c -lpthread
$ ./mt_webserver 30000 ./site
```

Visiting URL `http://localhost:30000/index.html`

Multithreaded programming

Main issue with writing multithreaded progams is that the threads execute in the *same address space*, so they share memory

A variable written by one thread may be read by another!

- Can be useful for communication between threads

- Can also be dangerous

Reentrancy

Some functions are designed to use global variables:

- strtok (for tokenizing C character string, retains state between calls)

- gethostbyname returns pointer to global struct hostent object

Reentrancy

Some functions are designed to use global variables:

- strtok (for tokenizing C character string, retains state between calls)

- gethostbyname returns pointer to global struct hostent object

Functions which use global variables are not *reentrant*

Some functions are designed to use global variables:

- strtok (for tokenizing C character string, retains state between calls)

- gethostbyname returns pointer to global struct hostent object

Functions which use global variables are not *reentrant*

''Reentrant'' means function can be safely ''reentered'' before a currently-executing call to the same function completes

Reentrancy

Some functions are designed to use global variables:

- strtok (for tokenizing C character string, retains state between calls)

- gethostbyname returns pointer to global struct hostent object

Functions which use global variables are not *reentrant*

''Reentrant'' means function can be safely ''reentered'' before a currently-executing call to the same function completes

Non-reentrant functions are dangerous for multithreaded programs (and also cause issues when called from recursive functions)

Tips for writing reentrant functions:

Tips for writing reentrant functions:

- Don't use global variables

Writing reentrant functions

Tips for writing reentrant functions:

- Don't use global variables

- Memory used by a reentrant function should be limited to

  - Local variables (on stack), or

Writing reentrant functions

Tips for writing reentrant functions:

- Don't use global variables

- Memory used by a reentrant function should be limited to

  – Local variables (on stack), or
  – Heap buffers not being used by other threads

Writing reentrant functions

Tips for writing reentrant functions:

- Don't use global variables

- Memory used by a reentrant function should be limited to

  - Local variables (on stack), or
  - Heap buffers not being used by other threads

- It's a good idea to have functions receive explicit pointers to memory they should use

Example: `strtok` vs. `strtok_r`

The strtok function uses an implicit global variable to keep
track of progress:

```
char buf = "foo bar baz";
printf("%s\n", strtok(buf, " "));     /* prints "foo" */
printf("%s\n", strtok(NULL, " "));    /* prints "bar" */
printf("%s\n", strtok(NULL, " "));    /* prints "baz" */
```

The strtok function uses an implicit global variable to keep track of progress:

```
char buf = "foo bar baz";
printf("%s\n", strtok(buf, " "));      /* prints "foo" */
printf("%s\n", strtok(NULL, " "));    /* prints "bar" */
printf("%s\n", strtok(NULL, " "));    /* prints "baz" */
```

The reentrant strtok_r function makes the progress variable explicit by taking a pointer to it as a parameter:

```
/* same output as code example above */
char buf = "foo bar baz", *save;
printf("%s\n", strtok_r(buf, " ", &save));
printf("%s\n", strtok_r(NULL, " ", &save));
printf("%s\n", strtok_r(NULL, " ", &save));
```

Example: `strtok` vs. `strtok_r`

The strtok function uses an implicit global variable to keep track of progress:

```
char buf = "foo bar baz";
printf("%s\n", strtok(buf, " "));     /* prints "foo" */
printf("%s\n", strtok(NULL, " "));    /* prints "bar" */
printf("%s\n", strtok(NULL, " "));    /* prints "baz" */
```

The reentrant strtok_r function makes the progress variable explicit by taking a pointer to it as a parameter:

```
/* same output as code example above */
char buf = "foo bar baz", *save;
printf("%s\n", strtok_r(buf, " ", &save));
printf("%s\n", strtok_r(NULL, " ", &save));
printf("%s\n", strtok_r(NULL, " ", &save));
```

Always use reentrant versions of library functions, and make your own functions reentrant!

For many (but not all!) multithreaded programs, it's useful to have explicit communication/interaction between threads

Concurrently-executing threads can use shared data structures to communicate

For many (but not all!) multithreaded programs, it's useful to have explicit communication/interaction between threads

Concurrently-executing threads can use shared data structures to communicate

But: concurrent modification of shared data is likely to lead to violated data structure invariants, corrupted program state, etc.

Synchronization

For many (but not all!) multithreaded programs, it's useful to have explicit communication/interaction between threads

Concurrently-executing threads can use shared data structures to communicate

But: concurrent modification of shared data is likely to lead to violated data structure invariants, corrupted program state, etc.

Synchronization mechanisms allow multiple threads to access shared data cooperatively

- More on this next time

- 10 second version: queues are awesome

Parallel computation

Computer Systems Fundamentals: Concurrency with pthreads

Assume C is a complex number, and $Z_0 = 0 + 0i$

Iterate the following equation an arbitrary number of times, starting with Z_0:

$$Z_{n+1} = {Z_n}^2 + C$$

Does the magnitude of Z ever reach 2 (for any finite number of iterations)?

- No $\rightarrow C$ is in the Mandelbrot set

- Yes $\rightarrow C$ is not in the Mandelbrot set

For some region of the complex plane, sample points and determine whether they are in the Mandelbrot set

Assume a point C is in the set if the equation can be iterated at large number of times without magnitude of Z reaching 2

For points C not in the set, choose a color based on number of iterations before magnitude of Z reaches 2

```
typedef struct { double real, imag; } Complex;

static inline Complex complex_add(Complex left, Complex right) {
  Complex sum = { left.real+right.real, left.imag+right.imag };
  return sum;
}

static inline Complex complex_mul(Complex left, Complex right) {
  double a = left.real, b = left.imag, c = right.real, d = right.imag;
  Complex prod = { a*c - b*d, b*c + a*d };
  return prod;
}

static inline double complex_mag(Complex c) {
  return sqrt(c.real*c.real + c.imag*c.imag);
}
```
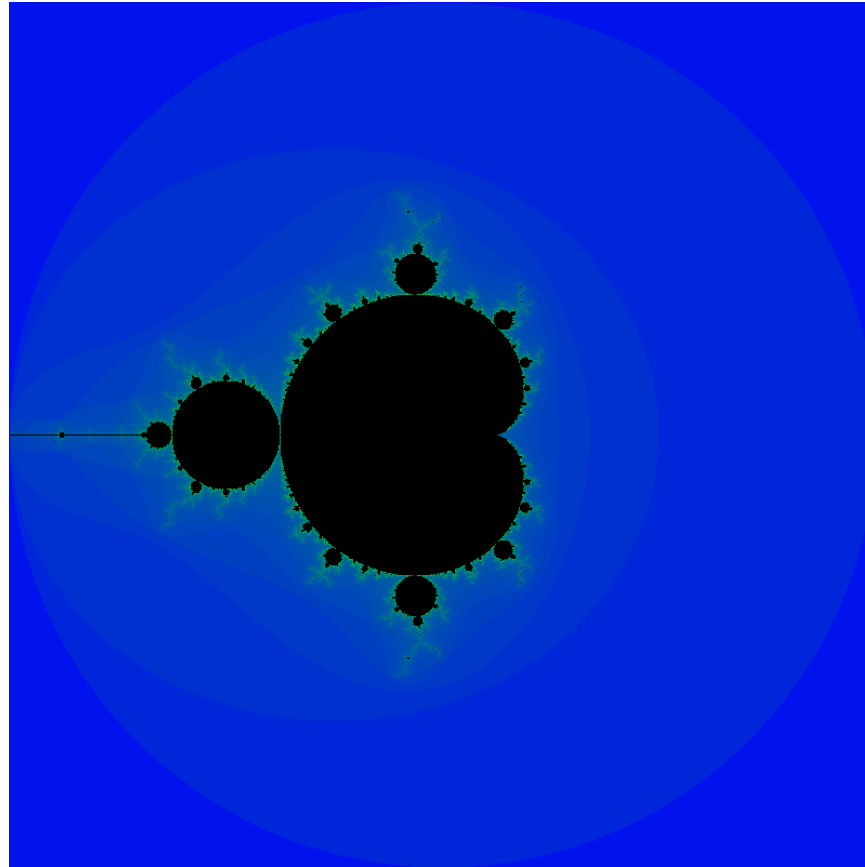
Function to iterate the equation for a specific complex number,
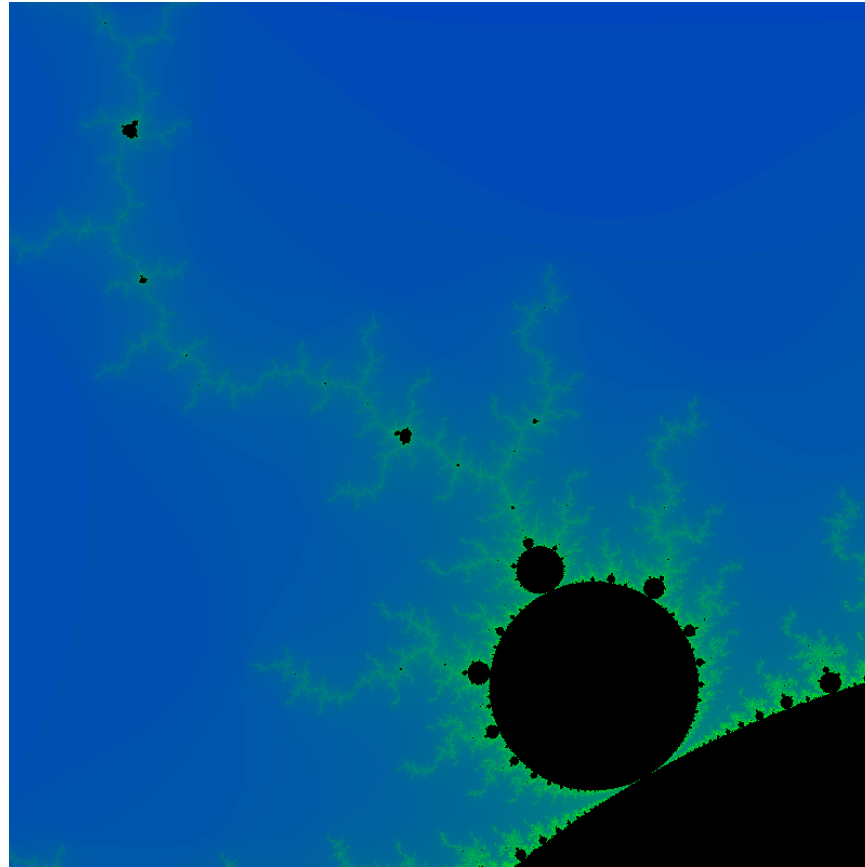up to a maximum number of iterations

```
int mandel_num_iters(Complex c) {
  Complex z = { 0.0, 0.0 };
  int num_iters = 0;
  while (complex_mag(z) < 2.0 && num_iters < MAX_ITERS) {
    z = complex_add(complex_mul(z, z), c);
    num_iters++;
  }
  return num_iters;
}
```

For complex numbers $a + bi$ where $-2 < a < 2$ and $-2 < b < 2$:

Visualization

For complex numbers $a + bi$ where $-1.28667 < a < -1.066667$ and $-0.413333 < b < -0.193333$:

Observation

The computation for each point in the complex plane is completely independent

- I.e., an *embarrassingly parallel* problem

We can speed up the computation by doing the computation for different points in parallel on multiple CPU cores

Approach:

- Use an array to store iteration counts (one per complex number)

- Create fixed number of computation threads

- Assign a subset of array elements to each computation thread

- When all threads have finished, use iteration counts to render image

Fork/join parallel computation

divide work

start computation threads

time

combine solutions

done
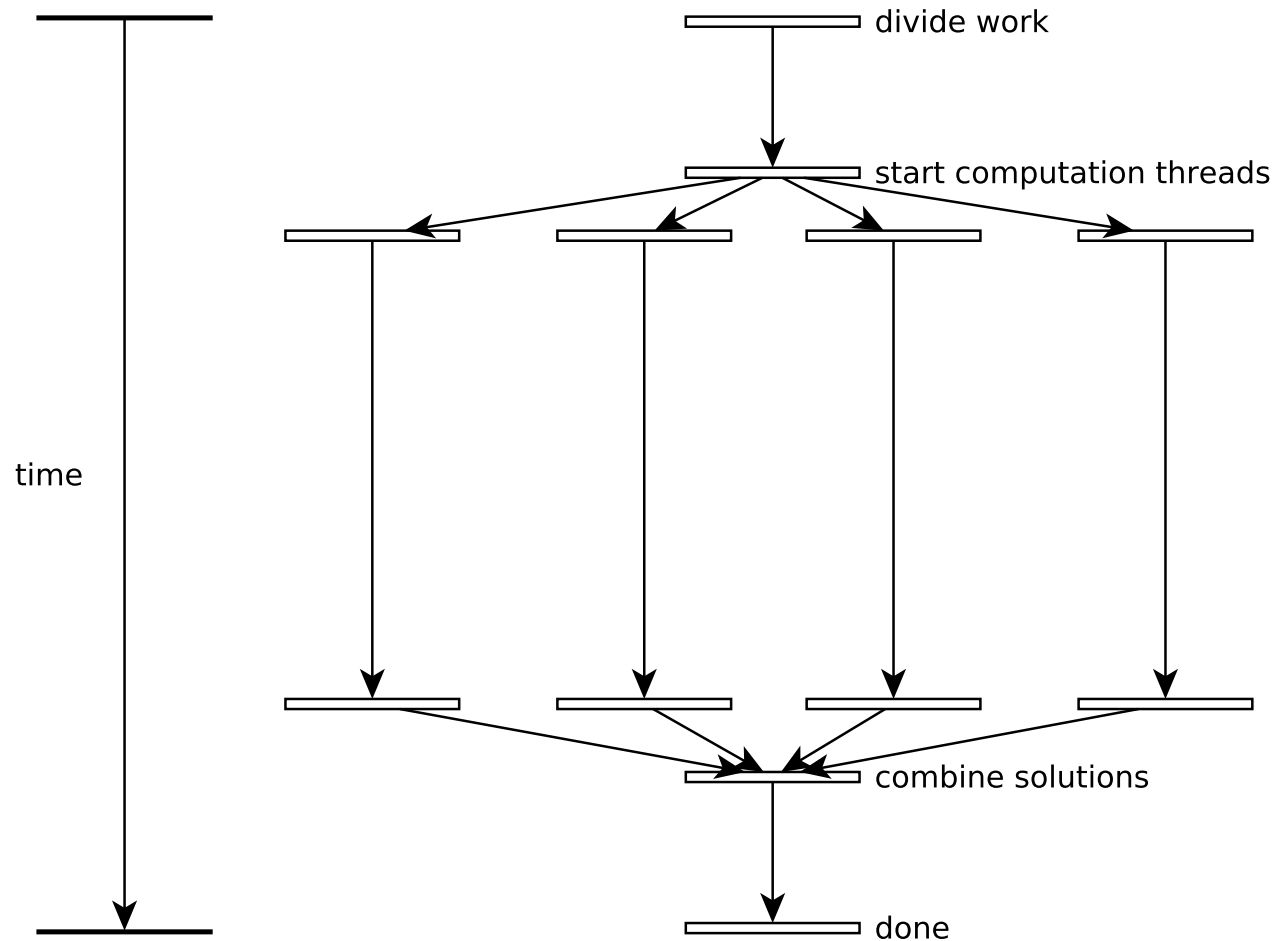
Sequential computation

Core of the sequential Mandelbrot computation:

```
int *iters = malloc(sizeof(int) * NROWS * NCOLS);
for (int i = 0; i < NROWS; i++) {
  mandel_compute_row(iters, NROWS, NCOLS,
    xmin, xmax, ymin, ymax,
    i);
}
```

The mandel_compute_row function computes iteration counts for
a row of complex numbers, storing them in the iters array

```
typedef struct {
  double xmin, xmax, ymin, ymax;
  int *iters;
  int start_row, skip;
} Work;

void *worker(void *arg) {
  Work *work = arg;

  for (int i = work->start_row; i < NROWS; i += work->skip) {
    mandel_compute_row(work->iters, NROWS, NCOLS,
      work->xmin, work->xmax, work->ymin, work->ymax,
      i);
  }

  return NULL;
}
```

```
/* master work assignment */
Work master = { xmin, xmax, ymin, ymax, iters, 0, NUM_THREADS };

/* start threads */
pthread_t threads[NUM_THREADS];
Work work[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; i++) {
  work[i] = master;
  work[i].start_row = i; /* each thread has different start row */
  pthread_create(&threads[i], NULL, worker, &work[i]);
}

/* wait for threads to complete */
for (int i = 0; i < NUM_THREADS; i++) {
  pthread_join(threads[i], NULL);
}
```

Results

Running sequential vs. 4 threads on Core i5-3470T (dual core, hyperthreaded):

```
$ time ./mandelbrot -1.286667 -1.066667 -0.413333 -0.193333
Success?

real 0m2.020s
user 0m2.012s
sys 0m0.008s
$ time ./mandelbrot_par -1.286667 -1.066667 -0.413333 -0.193333
Success?

real 0m0.815s
user 0m3.054s
sys 0m0.000s
```

Source code on web page: mandelbrot.zip