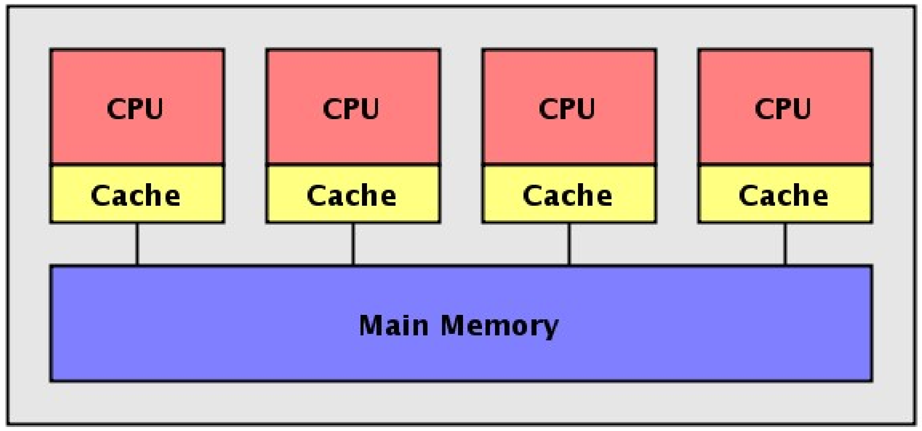
Shared Memory Programming and OpenMP

CS121 Parallel Computing

Fall 2023

Shared memory multiprocessor



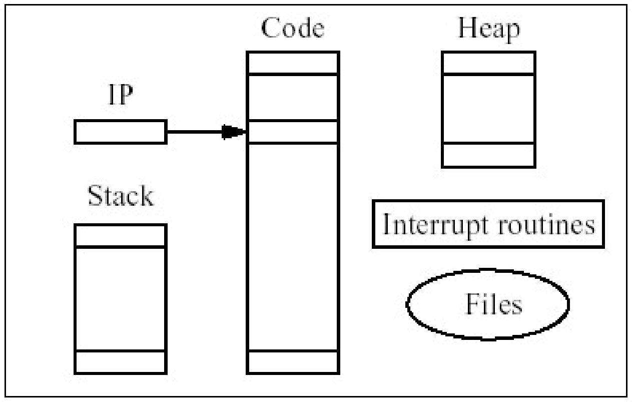
- Any memory location is accessible by any of the processors.
- A single address space exists.
  - Each memory location is given a unique address within a single range of addresses.
- Generally, more convenient than distributed memory programming.
  - But access to shared data needs to be controlled by the programmer, e.g. using critical sections.

Shared memory programming

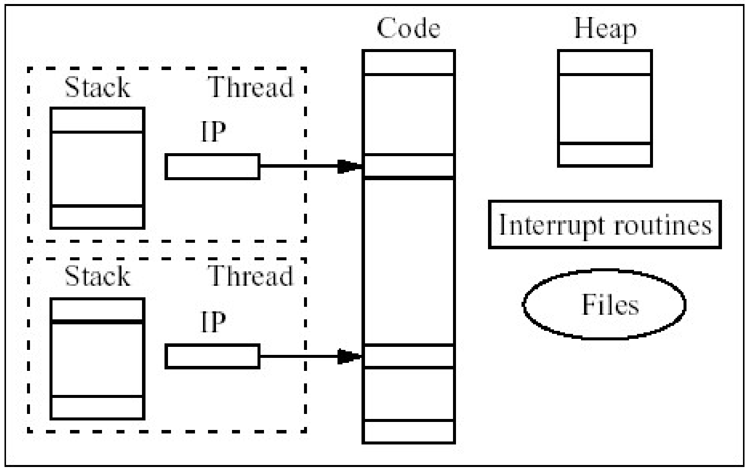
- Threads (e.g. Pthreads, Java)
  - The programmer decomposes the program into individual sequences of instructions (threads) that can execute in parallel and access shared data.
  - Very general, but hard to use because programmer must manage everything.
- Parallel programming language / library
  - A parallel language or library is used to create code that can be executed on a shared memory parallel architecture.
  - Requires new compiler, programmers to learn new language, etc.
- Compiler directives (e.g. OpenMP)
  - The programmer inserts compiler directives into a sequential program to specify parallelism and indicate shared data and the compiler translates into threads.
  - Still uses threads underneath, but system manages the threads.
  - Easy to program (though loses some flexibility). Requires less changes to compiler.
  - Most popular option.

Processes and threads

- Process (e.g. MPI)
  - Separate program with its own variables, memory, stack and instruction pointer.
  - Different programs can't access each other's memory.
- Thread (e.g. OpenMP)
  - Concurrent routine that shares the variables and memory space, but has its own stack and instruction pointer.



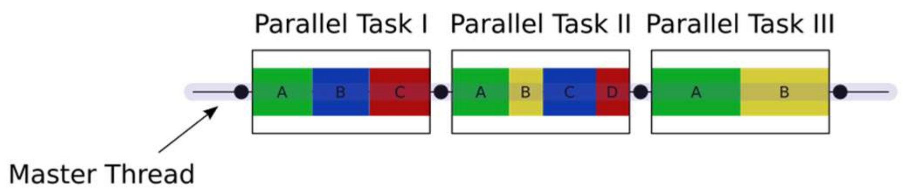
(a) Process

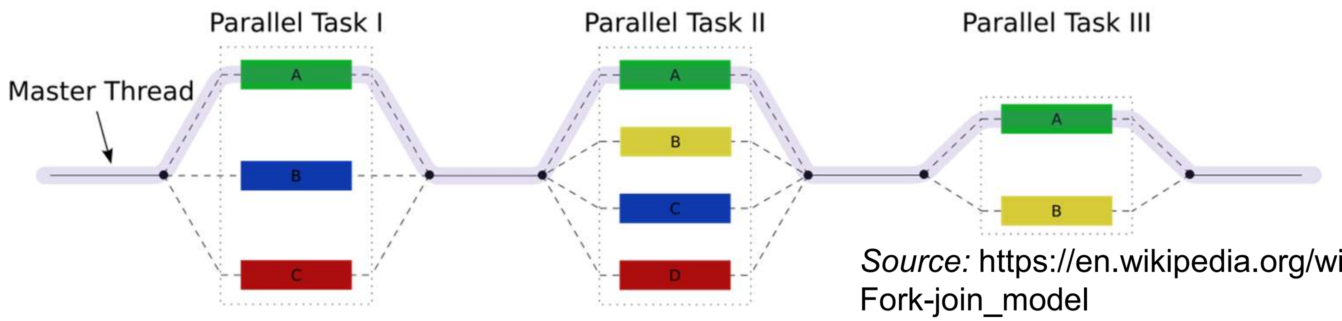


(b) Threads

Fork-join model

- A model for parallel computing using threads.
- Computation starts with master thread.
- If there is parallel work, master thread forks off slave threads.
  - Thread can be executed on same processor / core or a different one.
- When slave threads finish, they join (merge back into) master thread.

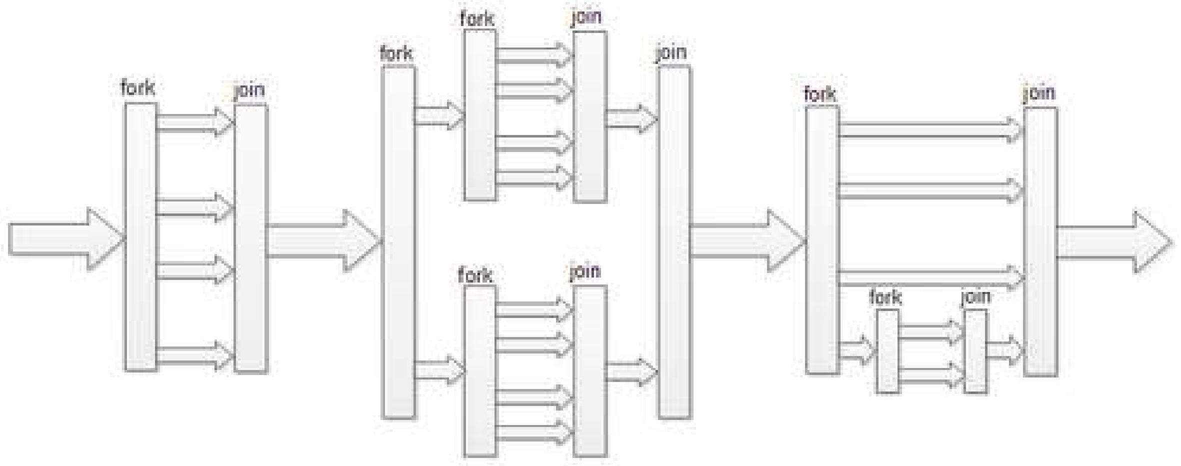




Source: https://en.wikipedia.org/wiki/Fork-join_model

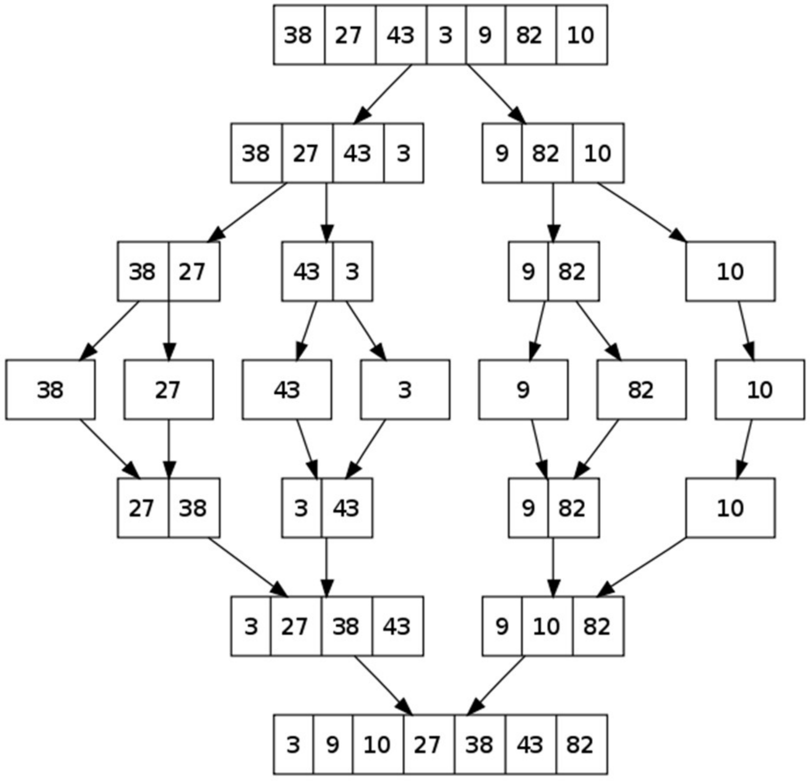
Fork-join model

- Spawned threads may recursively create further threads.
  - Slave threads join with the thread that spawned them.
  - Can also create detached threads, that don't do a join when they terminate.



Example

```
mergesort(A, lo, hi):
    if lo < hi:
        mid = ⌊(hi - lo) / 2⌋
         fork mergesort(A, lo, mid)
         mergesort(A, mid, hi)
         join
         merge(A, lo, mid, hi)
```



Source: https://en.wikipedia.org/wiki/Merge_sort

Statement execution order

- Single thread Execute statements in program order until blocked or end of time slice.
- Multi-threaded Instructions of different threads are interleaved in arbitrary order.
  - Correctness of program can't depend on particular interleaving order, or else race condition bug.
  - Ensuring no race conditions one of the primary challenges to shared memory programming.

Thread 1	**Thread 2**	**Possible interleaving**
Instruction 1.1	Instruction 2.1	Instruction 2.1
Instruction 1.2	Instruction 2.2	Instruction 1.1
Instruction 1.3	Instruction 2.3	Instruction 1.2
		Instruction 2.2
		Instruction 2.3
		Instruction 1.3

Race condition example

- Accessing shared data needs careful control because of interleaving of threads.
- Consider two threads which increment a shared counter x.
  - In sequential execution, x equals 2 at the end.
  - In parallel execution under given interleaving, x equals 1.

Thread 1	**Thread 2**	**Possible interleaving**
load x	load x	load x
compute x+1	compute x+1	compute x+1
store x	store x	load x
		store x
		compute x+1
		store x
		// x == 1

Thread safe routines

- A routine is thread safe if it can be called from multiple threads simultaneously and always produces correct results.
    - Standard I/O routines are thread safe.
        - Ex messages are printed without interleaving the characters.
    - Other system routines may not be thread safe, e.g. some random number generators
- Routines that access shared data may require special care to be made thread safe.
- If a routine is not thread safe, it must be executed by only one thread at a time in a “critical section”.

Critical sections

- A block of code that can be executed by only one thread at a time.
  - Multiple changes can be made to data without interruption, so that data transitions from safe state to safe state.
  - Also called mutual exclusion.
  - Also appears in operating systems and programming languages, e.g. Java's `synchronized` statement.
- Helps avoid the race condition bugs we saw earlier.

Thread 1	**Thread 2**	**Possible interleaving**
load x	load x	load x
compute x+1	compute x+1	compute x+1
store x	store x	store x
		load x
		compute x+1
		store x
		// x == 2

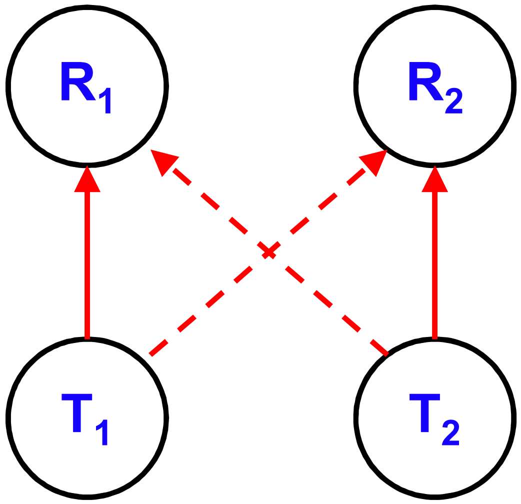
Locks

- A simple mechanism for ensuring mutual exclusion.
- A thread sets a lock before entering the critical section, and unsets it when it leaves.
- If a thread tries to set a lock and finds it locked, it blocks, i.e. waits for the lock to be unset.
  - So only the first thread to set the lock can execute the code in the critical section.
  - Other threads wait until the first thread finishes the critical section and unsets the lock, after which one of them can set the lock and perform the critical section.

```
set_lock(mutex);
critical section
...
unset_lock(mutex);
```

Deadlock

- A system state when all threads are stuck, i.e. can't take another step.
- Can occur when a thread T_1 waits for a resource held by T_2, while T_2 waits for a resource held by T_1.
- Can also have a waiting cycle of many threads.
- Can avoid deadlock by having all threads lock resources in same order.



Non-blocking locking

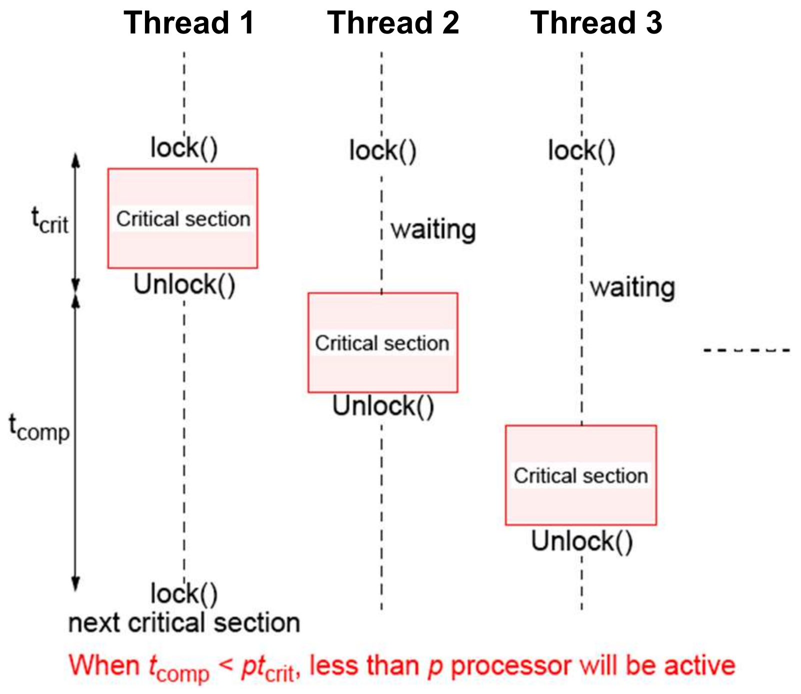
- Attempt to lock without blocking.

```
flag = test_lock(mutex);
if (!flag) {
    critical section
    unset_lock(mutex);
} else …
```

- If lock currently unset, it will set it and return success.
- If lock currently set, it will return failure without blocking.
- Can avoid deadlock.
  - Threads can use `test_lock` to access resource.
- Can avoid waiting time associated with blocking.
  - Thread can do other work and `test_lock` again later.

Critical sections and performance

- Critical sections lead to serialization of code.
  - If multiple threads want access to a critical section and reach it at the same time, the threads must be executed sequentially.
  - Then the execution time becomes almost that of a single processor.
- For performance, avoid critical sections when possible, and minimize their size.



Condition variables

- Often, a critical section needs to be executed only when a specific condition is met.
- Can use a condition variable.
  - Thread gets the lock for a critical section, then calls the condition variable to wait for condition to become true.
  - Waiting thread goes to sleep and releases lock, atomically.
  - If there are several waiters, they get put in a queue.
  - On a `signal_all`, one of the waking threads reacquires lock.
- More efficient than continually testing a lock to see when condition met.
- `wait(cond, lock)`
  - Atomically release `lock` and go to sleep. Upon waking, try to reacquire `lock`.
- `signal(cond)`
  - Wake up one sleeping thread waiting on `cond`.
- `signal_all(cond)`
  - Wake up all sleeping threads waiting on `cond`.

Producer-consumer example

- Producer threads add items to a queue, consumer threads remove them.
  - If queue empty, consumers wait. If queue full, producers wait.
- Instead of continuously locking the queue and checking if it's full / empty, go to sleep until signaled.
- Accesses to queue still need to be protected using `lock`.
- Producers and consumers signal each other using condition variables `not_full`, `not_empty`.
- Use `while` loop around `wait` because there may be multiple producers, consumers.
  - E.g. producer's `signal_all` can wake several consumers, one of which consumes the queue item. So the other consumers should check again whether items == 0.

```
Producer() {
   set(lock)
   while (items == N)
      wait(not_full, lock);
   /* access shared resource */
   items++;
   signal_all(not_empty);
   release(lock);
}
```

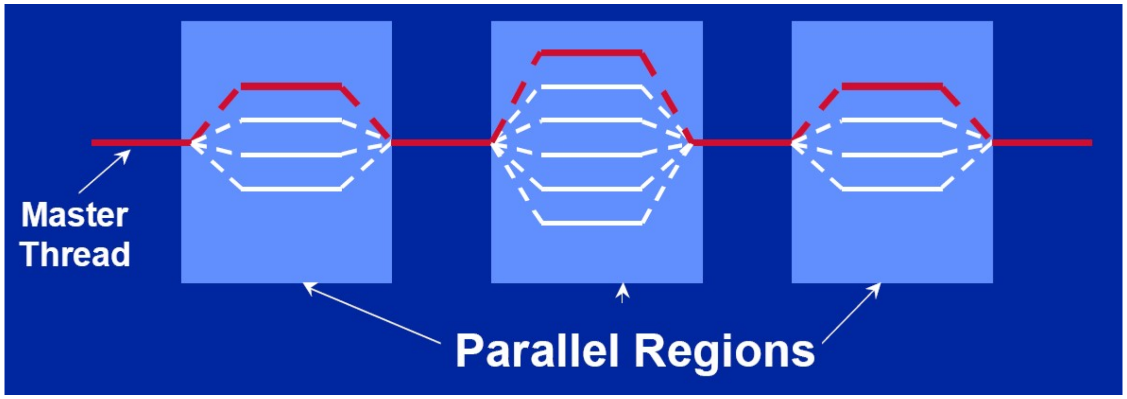
```
Consumer() {
   set(lock)
   while (items == 0)
      wait(not_empty, lock);
   /* access shared resource */
   items--;
   signal_all(not_full);
   release(lock);
}
```

OpenMP

- OpenMP is a standard for shared memory programming adopted by many hardware vendors.
- Can be used with different languages, e.g. C, C++ and Fortran.
- Compiler directives are used to specify parallelism and to indicate shared data.
- An OpenMP compatible compiler produces parallel program using the directives. A noncompatible compiler produces correct sequential program using same code.
  - Several OpenMP compilers available, e.g. Intel C compiler.
- Can be used to add parallelism incrementally to a sequential program, e.g. by parallelizing for loops.
- Underneath, OpenMP still uses threads.
  - OpenMP gives a more convenient, succinct way to manage threads.
  - But it lacks some of the expressiveness of explicit threading.

OpenMP

- OpenMP is based on threads, and uses the "fork-join" model.
  - Initially, a single master thread exists.
  - Parallel regions (sections of code) can be executed by a team of threads.
  - Compiler takes care of creating and coordinating threads.
- Available for C / C++ and Fortran. Documentation at http://openmp.org/wp/openmp-specifications/



Parallel regions

- The parallel directive forks a team of threads, each of which executes the following region, enclosed in {...}.

```
#pragma omp parallel
structured-block // { ... code ... }
```

- Threads do a join at end of parallel region, and execution resumes with the single master thread.
- Number of threads can be set by
  - num_threads clause after the parallel directive.
  - omp_set_num_threads() library routine previously called.
  - Environment variable OMP_NUM_THREADS.
  - Recommendation is one thread per processor / core.
- Threads can do the work in the region in parallel.
  - Can do different things based on thread ID.
  - Can share work using for, sections, task, etc. directives.
- Parallel regions can be nested.

Parallel regions

- Example

```
#pragma omp parallel private(iam, np)
{
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d\n",
            iam, np);
}
```

  - All threads in parallel region run this code.
  - `iam` and `np` are private variables (i.e. instance of variable for each thread).
  - `omp_get_num_threads()` returns the number of threads n in the team used for the parallel region.
  - `omp_get_thread_num()` returns thread number (identity) in range 0 to n-1 with master thread 0.
  - Messages printed in arbitrary order.

Work sharing

- Share some work inside a parallel region among threads.
- For example, `for` construct inside a parallel region partitions iterations of the loop among the threads.

```
#pragma omp for
for(i=0; i<n; i++)
    {do_stuff(i);}
```

  - The way in which iterations are assigned to threads can be specified by an additional `schedule` clause.
- For this and other worksharing constructs:
  - Does not start a new team of threads - that is done by an enclosing `parallel` construct.
  - Implicit barrier at the end of the construct unless a `nowait` clause is included. I.e. each thread will wait at end of construct for all other threads to finish.

Schedule clause

- Used for assigning iterations of parallel `for` to threads.
- `schedule(static[,chunk])`
  - Each thread gets a chunk of iterations of size “chunk” – by default chunks approximately equal.
  - Chunks assigned in round robin order.
- `schedule(dynamic[,chunk])`
  - Each time a thread finishes its iterations, grabs “chunks” more iterations, until all have been executed – default is 1.
  - Dynamic scheduling has some overhead, but can result in better load balancing if iterations not all equal sized.
- `schedule(guided[,chunk])`
  - Each thread dynamically grabs iterations where the size starts large and shrinks down to “chunk”.
  - Dynamic load balancing with less overhead.
- `schedule(runtime)`
  - Schedule type and chunk size taken from the OMP_SCHEDULE environment variable.

Combined `parallel for`

- If a `parallel` directive is followed by a single for directive, they can be combined.

```
#pragma omp parallel for schedule(static)
for (i=0; i<n; i++) { a[i] = a[i] + b[i];}
```

- Several restrictions on structure of for loop.
  - Number of iterations n must not change.
  - Loop increment must be fixed.
  - Must not exit loop prematurely (with `break, goto, throw`).
  - Purpose of restrictions is so amount of work in loop can be determined at start.

Different ways to parallelize for

```
// sequential

for (i=0; i<N; i++) {
     a[i] = a[i] + b[i];
}
```

```
// create parallel region
// then do worksharing

#pragma omp parallel {
     #pragma omp for
     for (i = 0; i < N; i++) {
          a[i] = a[i] + b[i];
     }
}
```

```
// create parallel region and do
//worksharing together

#pragma omp parallel for schedule(static)
     for (i = 0; i < N; i++) {
          a[i] = a[i] + b[i];
     }
```

```
// manual parallelization

#pragma omp parallel {
     int id, i, Nthreads, start, end;
     id = omp_get_thread_num();
     Nthreads = omp_get_num_threads();
     start = id * N / Nthreads;
     end = (id + 1) * N / Nthreads;
     for (i = start; i < end; i++) {
          a[i] = a[i] + b[i];
     }
}
```

```
// threads do redundant work

#pragma omp parallel {
     for (i = 0; i < N; i++) {
          a[i] = a[i] + b[i];
     }
}
```

Other work sharing constructs

- Sections construct
  - Each thread assigned some sections of work.
  - Threads can be assigned 0, or multiple sections of work.
  - There's an implicit barrier at end of sections block, i.e. threads wait for each other to finish all sections before executing code after section.
  - Can turn off barrier using `nowait`.

```
#pragma omp parallel {
    #pragma omp sections {
        #pragma omp section
        { // do stuff }
        #pragma omp section
         { // do stuff }
        ...
    }
}
```

Other work sharing constructs

- Single construct
  - Structured block is executed by one thread of parallel region only (not necessarily master thread).
  - Barrier implied unless use `nowait`.
  - For doing tasks that should only be done by one thread when inside a parallel region.
- Master construct
  - Structured block is executed by master thread only. No implicit barrier at end.

```
#pragma omp parallel {
    #pragma omp single {
    // do stuff
    }
}
```

```
#pragma omp parallel {
    #pragma omp master {
    // do stuff
    }
}
```

Data environment

- OpenMP has a shared memory programming model.
  - Some variables are shared and accessible by all threads.
  - Other threads are private, and each thread has its own copy.
- Most variables are shared by default.
  - Global and static variables are shared.
  - Variables declared in master thread shared by default.
- Some variables parallel blocks private by default.
  - Loop index of `for` / `parallel for` construct.
  - Stack variables (e.g. function argument or local variable) created during execution of a `parallel` region.
  - Automatic variables in functions called in `parallel` region.

Data environment

- Variable status can be changed in `parallel` regions and worksharing constructs, except `shared` which only applies to `parallel` regions.
  - `shared(variable-list)`
  - `private(variable-list)`
- Can also add `default(private)` or `default(shared)` clause to make shared variables private or shared by default.

```
1 int x = 5;
2 #pragma omp parallel private(x) {
3   int p = omp_get_thread_num();
4    x = p;
5   printf(“private x is %d\n",x);
6 }
7 printf(“shared x is %d\n",x);
```

- At line 1, x is shared.
- At line 3, each thread has a private copy of x, but x’s value is uninitialized.
- At line 5, every thread prints a different x.
- At line 7, master thread prints x is 5.

Data environment

- When entering parallel region, set the initial values of private variables to be its value outside region using `firstprivate(variable-list)`
- When exiting `parallel for`, set the values of private variables outside the region to be their values in the final iteration of the `for` loop using `lastprivate(variable-list)`

```
int tmp = 2;
#pragma omp parallel for firstprivate(tmp) lastprivate(tmp)
for (int i = 0; i < 10; i++)
// each thread has a private tmp initialized to 2
    tmp++;
}
// prints a value for tmp != 2; the value depends on which
// thread performed the last iteration of the for loop, and
// how many iterations that loop performed
printf("%d\n", tmp);
```

Data environment

- Reduction combines values from threads.
  - `reduction(op : variable-list)`
    - Variables in the list must be shared in the enclosing `parallel` region.
    - Each thread initially makes a local copy of each list variable and updates it.
    - Local copies are reduced into a single global copy at the end of the construct.
    - More efficient than using a critical section.

```
#pragma omp parallel for reduction (+ : x)
for (i=0; i<n; i++) {
    x = x + a[i]; }
```

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    #pragma omp critical
     {x = x + a[i];}}
```

Synchronization constructs

- OpenMP has critical sections and locks to protect accesses.
- Critical sections

  `#pragma omp critical [name] structured-block`
  - Only one thread can execute associated structured block at a time.
  - Name can be used to identify the critical section. Critical sections with no name default to the same.
- Locks

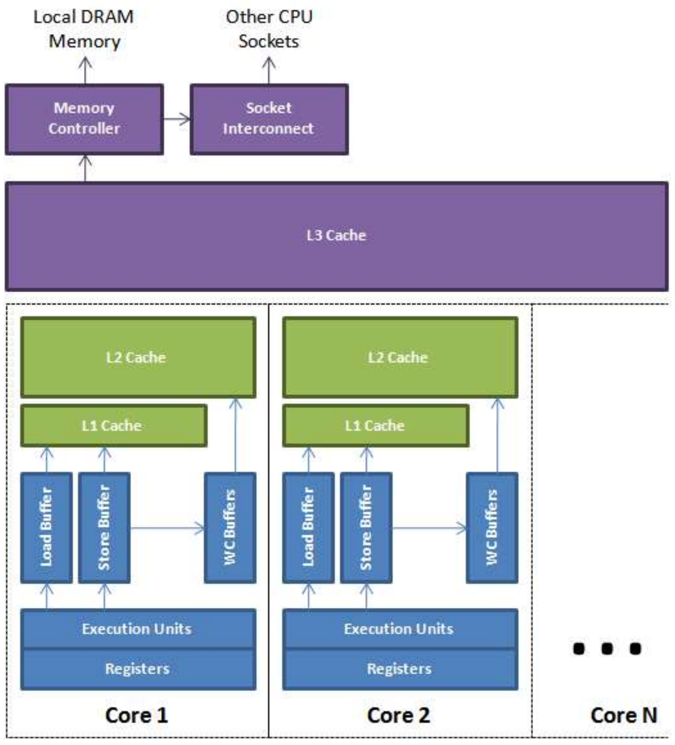
  `omp_init_lock(arg), omp_set_lock(arg), omp_unset_lock(arg), omp_test_lock(arg), omp_destroy_lock(arg)`
  - `arg` is a memory location.
- Critical sections protect sections of code, but locks protect data.
  - Ex Consider a hash function insert routine.
    - A critical section around the routine allows only one thread to insert at a time, even when different threads want to insert to different locations.
    - We only want to prevent concurrent inserts to same table entry. So associate one lock with each table entry.
- Barriers

  `#pragma omp barrier`

  All threads must reach the barrier before any can proceed.

Synchronization constructs

- Atomic operations `#pragma omp atomic expression-statement`
  - Only one thread can execute the associated `expression-statement` at a time.
  - Only works for simple statements such as x++, max, test&set, etc.
  - Done in hardware; more efficient than locks or critical sections.
- Flushing values
  - `#pragma omp flush [(var)]`
  - Writes listed variables from buffer to cache or memory to ensure all processors observe latest variable values.



Synchronization constructs

- Ordered statements are used in `for` and `parallel for` constructs to cause the subsequent structured block to be executed in strict loop order.
  - Code outside the ordered block can still execute in parallel.
- Should usually use static schedule with small chunk size.

```
#pragma omp parallel for ordered
   schedule(static, 1)
for (i = 0; i < n; i += 1)
     // do stuff in interleaved order
     // s, t are increased / decreased in
     // order 0, 1, 2, ...
     #pragma omp ordered {
          s += i;
          t -= i;
     }
```

Static schedule with default chunk size

tid	List of iterations	Timeline
0	0,1,2	==o==o==o
1	3,4,5	==.......o==o==o
2	6,7,8	==..............o==o==o

Static schedule with chunk size 1

tid	List of iterations	Timeline
0	0,3,6	==o==o==o
1	1,4,7	==.o==o==o
2	2,5,8	==..o==o==o

Runtime execution

- Runtime environment routines
  - Number of threads

    `omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num()`
  - Number of processors

    `omp_num_procs(),`
  - Currently in active region?

    `omp_in_parallel()`
  - Allows number of threads in parallel regions to be adjusted dynamically

    `omp_set_dynamic(int), omp_get_dynamic()`

OpenMP example

- Mandelbrot Set

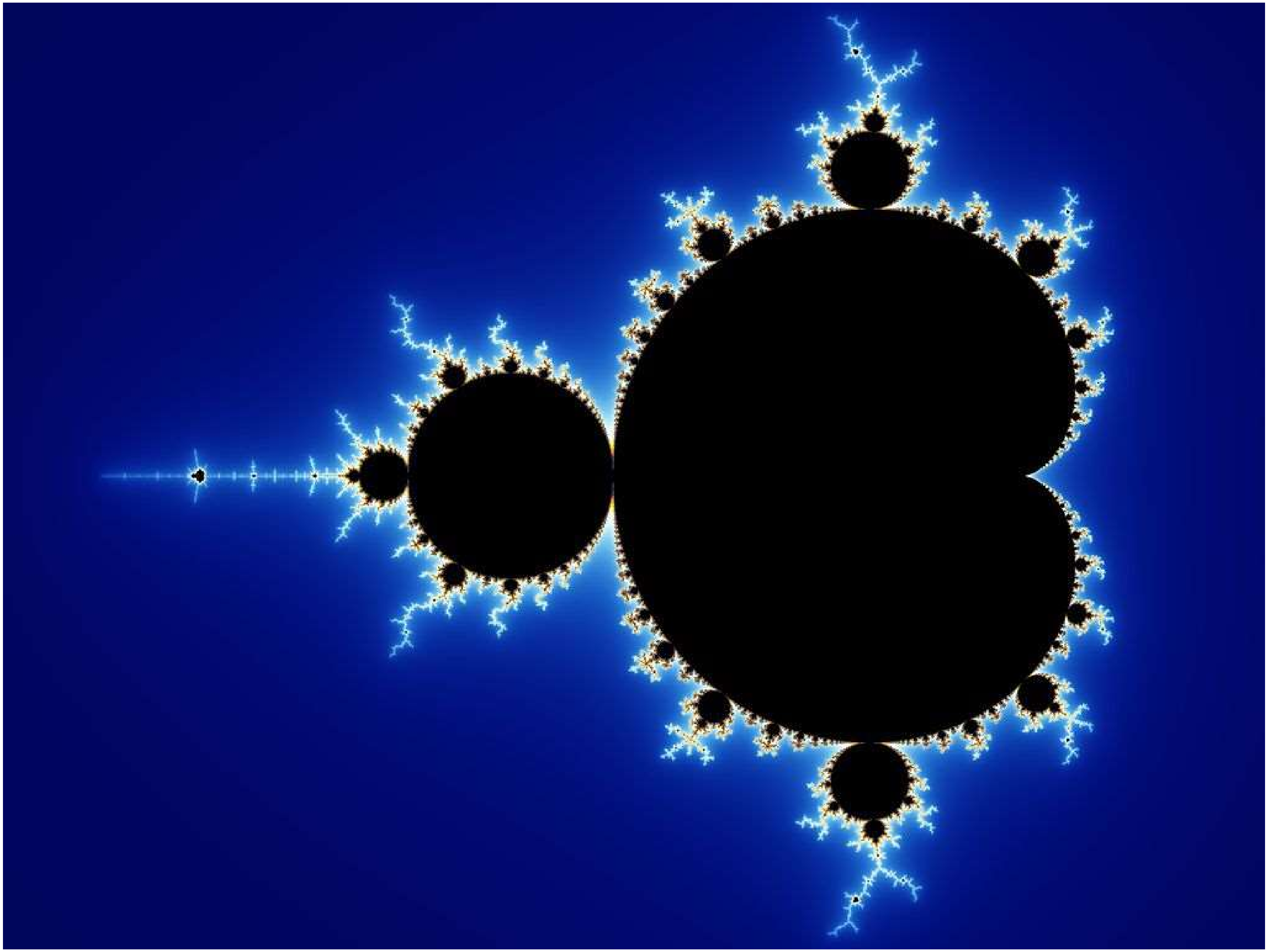
Set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function

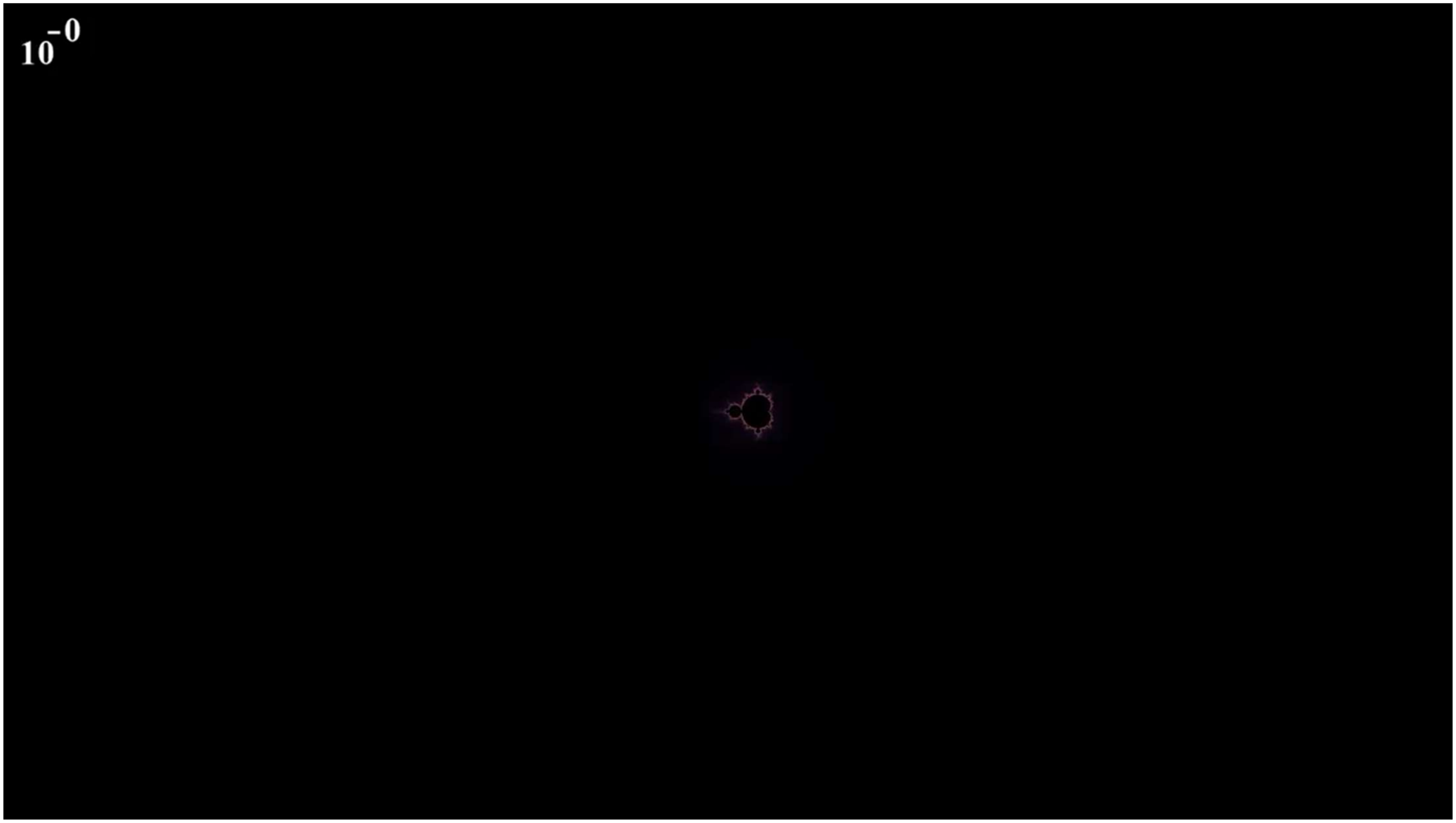
$$z_{k+1} = z_k^2 + c$$

where z_{k+1} is the $(k + 1)$th iteration of the complex number $z = a + bi$ and c is a complex number giving position of point in the complex plane. The initial value for z is zero.

Iterations continued until magnitude of z is greater than 2 or number of iterations reaches arbitrary limit. Magnitude of z is the length of the vector given by

$$z_{\text{length}} = \sqrt{a^2 + b^2}$$




10^{-0}

Sequential routine

```
structure complex {
    float real;
    float imag;
};

int calpixel(complex c) {
    int count, max;
    complex z;
    float temp, lengthsq;
    max = 256;
    z.real = 0; z.imag = 0;
    count = 0; /* number of iterations */
    do {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        count++;
    } while ((lengthsq < 4.0) && (count < max));
    return count;
}
```

$z_{k+1} = z_k^2 + c$

$z^2 = (a + bi)*(a + bi)$
$\quad = a^2 - b^2 + 2abi$

count gives colour (or intensity) to be displayed

It's known z will diverge if $|z| \geq 2$.

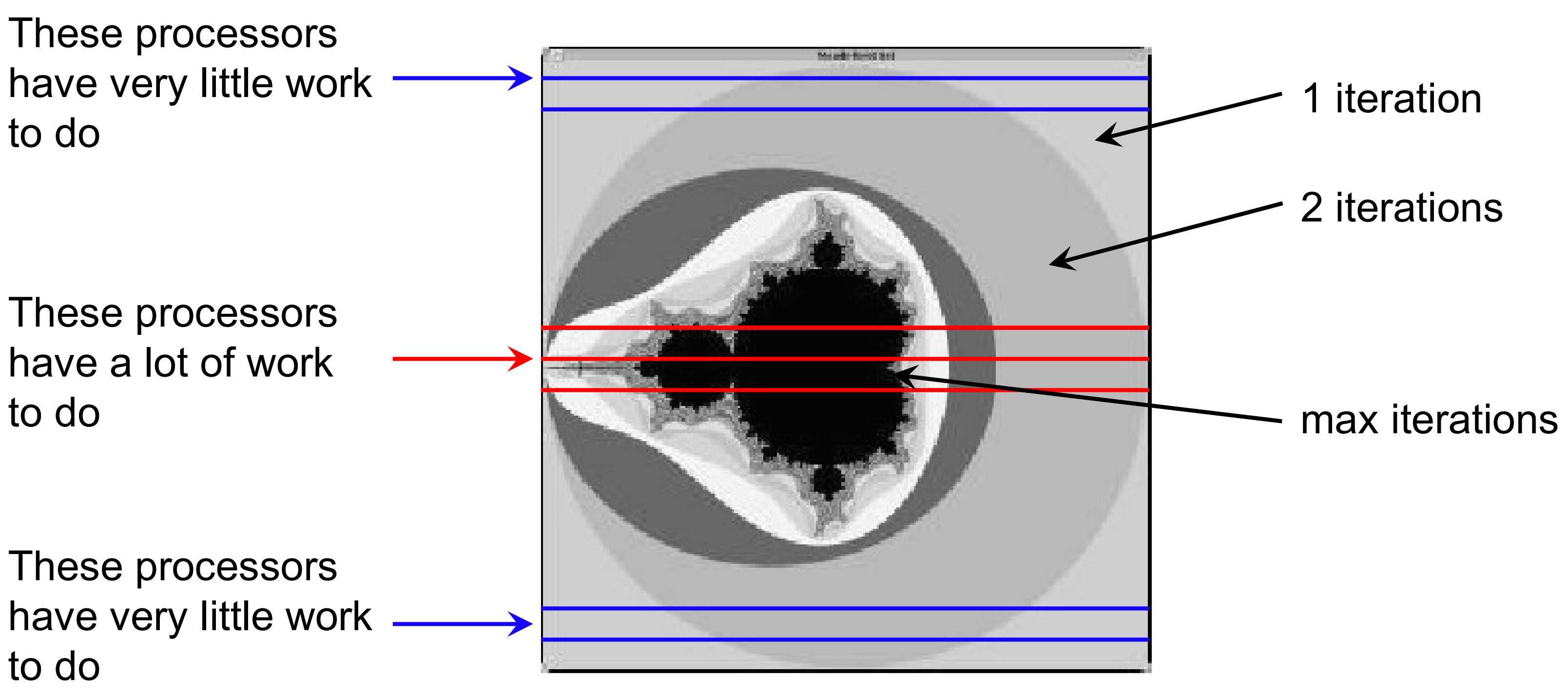
Parallelization of Mandelbrot

- Calculations for each pixel are independent.
  - Sometimes called an embarrassingly parallel computation.
- Static assignment
  - Divide the image into groups of pixels by row and assign each group to a separate thread.
  - By default, group (chunk) size is approximately equal.

```
#pragma omp parallel for private(j) schedule (static)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        colour[i][j] = calpixel(i,j);
```

  - Not efficient as different pixels require different numbers of iterations and the computation time of different strips will vary considerably.

Static schedule


These processors have very little work to do
These processors have a lot of work to do
These processors have very little work to do
1 iteration
2 iterations
max iterations

- This is a load balancing problem. Processors for top and bottom rows mostly idle, while processors for middle rows have lots of computation.

Parallelization of Mandelbrot

- Cyclic assignment

```
#pragma omp parallel for private(j) schedule (static, 1)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        colour[i][j] = calpixel(i,j);
```

  - Iterations are assigned in a round robin manner.
  - Each thread receives a mixed set of tasks, some with a lot, some with little computation.

- Dynamic assignment

```
#pragma omp parallel for private(j) schedule (dynamic, 1)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
         colour[i][j] = calpixel(i,j);
```

  - When a thread has finished the current row, it receives a new row to compute.
  - Can also use guided.

PGAS languages

- Partitioned Global Address Space is another model for thread based shared memory parallelism.
- Includes a number of languages, e.g. Unified Parallel C (UPC), Coarray Fortran (CAF), Global Arrays, etc.
  - These are based on loop parallelism, like OpenMP.
- Also asynchronous PGAS languages, e.g. X10 and Chapel.
  - Parent threads explicitly spawn and synch with child threads.
- So far not very widely used, and still requires tuning for good performance.

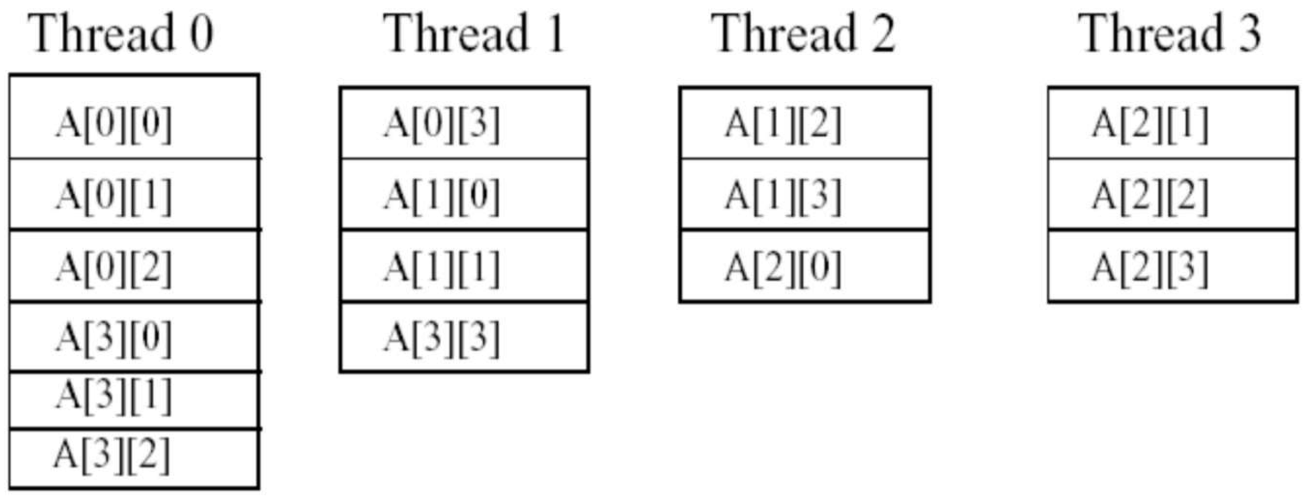
PGAS memory model

- A global address space accessible to all threads.
- However, address space is divided into partitions, and each thread has an affinity to one partition.
  - This partition is the local memory of a processor. The other partitions are local memories at other processors.
  - Thread also has private data only it can access.
- The convenience of OpenMP, but a more precise performance model because it captures data locality.
- Supports pointers to shared and private data, and static and dynamic memory allocation.

PGAS arrays

- Arrays can be partitioned across threads to increase local memory accesses and performance.
- Ex Assume THREADS = 4. In UPC A array is distributed as follows among the threads' memories.
    - Allocate in blocks of 3, row major order, round robin through threads.

```
shared [3] int A[4][THREADS]
```



UPC constructs

- For loops are parallelized with `upc_forall(init; test; update; affinity)`
  - `Affinity controls which threads execute which iterations.`

```
shared double x[N], y[N], z[N];
int main() {
     int i;
     upc_forall(i=0; i < N; ++i; i)
          z[i] = x[i] + y[i];
}
```

- Expressions for barrier synchronization and locks.
- Synchronize memory between threads using fences and strict / relaxed memory models.

Other methods for shared memory

- MPI 2 and 3 support one sided communication, allowing processes to directly read or write data from each other without passing messages.
- Can also combine MPI and OpenMP.
  - Use MPI between different nodes, and OpenMP within each node.
  - OpenMP can improve load balancing and reduce number of small messages, both weaknesses of MPI.
  - Must use MPI library that supports threads.