Contents

OpenMP

- Language extension for C/C++
- Uses `#pragma` feature
  - Pre-processor directive
  - Ignored if the compiler doesn't understand
- `#include <omp.h>`
- `gcc -fopenmp progam.c`
- Has become an important standard for parallel programming

  - Makes the simple cases easy to program
  - Avoids thread-level programming for simple parallel code patterns
- We are looking at OpenMP because
  - Its a nice example of a small domain-specific language
  - Shows how otherwise difficult problems can be solved by sharply reducing the space of problems
- It was originally designed for expressing thread parallelism on counted loops
  - Aimed at matrix computations
- OpenMP 3 added "tasks"
  - Exploiting thread parallelism of
    * Non counted while loops
    * Recursive divide and conquer parallelism
- OpenMP 4 added SIMD and "accelerators"
  - Vector SIMD loops
  - Offload computation to GPUs

Threading Model

- OpenMP is all about threads
  - Or at least the core of OpenMP up to version 3
- There are several threads
  - Usually corresponding to number of available processors
  - Number of threads is set by the system the program is running on, not the programmer
    * Your program should work with any number of threads
- There is one master thread
  - Does most of the sequential work of the program
  - Other threads are activated for parallel sections

```
#pragma omp parallel
{
    x++;
}
```

- The same thing is done by all threads

- All data is shared between all threads
- Value of `x` at end of loop depends on
  - Number of threads
  - Which order they execute in
- This code is non-deterministic and will produce different results on different runs
- We rarely want all the threads to do exactly the same thing
- Usually want to divide up work between threads
- Three basic constructs for dividing work
  - Parallel for
  - Parallel sections
  - Parallel task

Parallel For

- Divides the iterations of a for loop between threads

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

- All variables shared
- Except loop control variable
- The iterations of the loop are divided among the threads
- Implicit barrier at the end of the for loop
  - All threads must wait until all iterations of the for loop have completed

Conditions

- Loop variable must be of type integer
- Loop condition must be of the form `i <, <=, >` or `>= loop_invariant_integer`
- A loop invariant integer is an integer whose value doesn't change throughout the running of the loop
- The third part of the for loop must be either an integer addition or subtraction of the loop variable by a loop invariant value
- The loop must be a single entry and single exit loop, with no jumps from the inside out or from the outside in

Parallel Sections

- Parallel *for* divides the work of a for loop among threads
  - All threads do the same for loop, but different iterations
- Parallel *sections* allow different things to be done by different threads
  - Allow unrelated but independent tasks to be done in parallel

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        min = find_min(a);
    }
    #pragma omp section
    {
        max = find_max(a);
    }
}
```

- Parallel sections can be used to express independent tasks that are difficult to express with parallel for
- Number of parallel sections is fixed in the code
  - Although the number of threads depends on the machine the programming is running on

Parallel Task

- Need constructs to deal with
  - Loops where number of iterations is not known
  - Recursive algorithms
- Tasks are parallel jobs that can be done in any order
  - They are added to a pool of tasks and executed when the system is ready

```
#pragma omp parallel
{
    #pragma omp single // just one thread does this bit
    {
        #pragma omp task
        {
```

```
            printf("Hello ");
        }
        #pragma omp task
        {
            printf("world");
        }
    }
}
```

- Creates a pool of work to be done
- Pool is initially empty
- A task is added to the pool each time we enter a "task" pragma
- The threads remove work from the queue and execute the tasks
- The queue is disbanded when
  - All enqueued work is complete
  - End of parallel region
  - Explicit `#pragma omp taskwait`
- Tasks are very flexible
  - Can be used for all sorts of problems that don't fit well into parallel for the parallel sections
  - Don't need to know how many tasks there will be at the time we enter the loop
- But there is an overhead of managing the pool
- Order of execution is not guaranteed
  - Tasks are taken from pool in arbitrary order whenever a thread is free
- But *it is* possible for one tasks to create another
  - Allows a partial ordering of tasks
  - If task A creates task B, then we are guaranteed that task B starts after task A

Scope of Data

- By default, all data is shared
- This is okay if the data is not updated
- A really big problem if multiple threads update the same data
- Two solutions
  - Provide mutual exclusion for shared data
  - Create private copies of data

Mutual Exclusion

- Mutual exclusion means that only one thread can access something at a time
- OpenMP provides two mechanisms for doing this
  - Atomic updates
  - Critical sections

Atomic Updates

- An atomic update can update a variable in a single unbreakable step

```
#pragma omp parallel
{
    #pragma omp atomic
    x++;
}
```

- In this code we are guaranteed that `x` will be increased by exactly the number of threads
- Only certain operators can be used in atomic updates
  - `x++, ++x, x--, --x`
  - `x op= expr`
    - `* + - * / & ^ | << >>`
- Otherwise the update cannot be atomic

Critical Section

- A section of code that only one thread can be in at a time
- Although all threads execute same code, this bit of code can be executed by only one thread at a time

```
#pragma omp parallel
{
    #pragma omp critical
    {
        x++;
    }
}
```

- In this code we are guaranteed that `x` will be increased by exactly the number of threads

```
#pragma omp critical (update _x)
```

- Critical sections are much more flexible that atomic updates
- Everything you can do with atomic updates can be done with a critical section
- But atomic updates are
  - Faster than critical sections
    - Or at least they *might* be faster, depending on how they are implemented by the compiler
  - Less error prone

Private Variables

- By default all variables are shared
- But private variables can also be created
- Some variables are private by default
  - Variables declared within the parallel block
  - Local variables of function called from within the parallel block
  - The loop control variable in parallel for

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

- Each thread has its own copy of `local_sum` but another variable of the same name also exists outside the parallel region
- Strange semantics with private variables
  - Declaring variables private creates new variable that is local to each thread
  - No connection between this local variable and the other variable outside
    - It's *almost* like created a new local variable within a C/C++ block

  - Local variable is given default value (usually 0)
  - Value of "outside" version of the private variable is undefined after parallel region

Shared Variables

- By default all variables in a parallel region are shared
- Can also explicitly declare them to be shared
- Can opt to force all variables to be declared shared or non-shared
- Use `default(none)` declaration to specify this

Reductions

- A reduction involves combining a whole bunch of values into a single value
  - E.g. summing a sequence of numbers
- Reductions are very common operation
- Reductions are inherently parallel
- With enough parallel resources you can do a reduction in `O(log n)` time using a reduction tree

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

- A private copy of the reduction variable is created for each thread
- OpenMP automatically combined the local copies together to create a final value at the end of the parallel section
- Reduction can be done with several different operators
  - `+ - * & | ^ && ||`
- Using a reduction is simpler than dividing work between the threads and combining the result yourself
- Using a reduction is potentially more efficient

Scheduling Parallel For Loops

- Usually with parallel for loops, the amount of work in each iteration is roughly the same
  - Therefore iterations of the loop are divided approximately evenly between threads
- Sometimes the work in each iteration can vary significantly
  - Some iterations take much more time
  - Some threads take much more time
  - Remaining threads are idle
- This is known as poor "load balancing"
- OpenMP provides three scheduling options
  - static
    - Iterations are divided equally (default)
  - dynamic
    - Iterations are put onto a work queue, and threads take iterations from the queue whenever they become idle
    - You can specify a chunk size so that iterations are taken from the queue in chunks by the threads
  - guided
    - Similar to dynamic, but initially the chunk size is large, and as the loop progresses the chunk size becomes smaller
    - Allows finer grain load balancing towards the end

Conditional Parallelism

- OpenMP directives can be made conditional on runtime conditions
- There is a significant cost for executing OpenMP parallel constructs
- Conditional parallelism can be used to avoid this cost where the amount of work is small

```
#pragma omp parallel for if ( n > 128 )
for ( i = 0; i < n; i++ )
    a[i] = b[i] + c[i];
```

Caching and Sharing

- Shared variables are shared among all threads

- Copies of these variables are likely to be stored in the level 1 cache of each processor core
- If you write to the same variable from different threads then the contents of the different L1 caches need to be synchronised in some way
  - Cache coherency logic detects that the cache line is invalid because variable has been updated elsewhere
  - Need to load the cache line again
  - Expensive
- Sometimes called thrashing or "ping ponging" cache lines
- Should avoid modifying shared variables a lot
- Multiple threads do not have to modify the same variable to cause these sorts of performance problems with invalidation of shared cache lines
- You just need to share the same cache line, not necessarily the same data
- So you need to be careful about multiple threads

Vector SIMD

- OpenMP is a very popular language for writing parallel code and from time to time, there is a new version with new features
- OpenMP 4.0 added new constructs to direct the compiler to vectorise a loop

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

- The semantic of OpenMP SIMD are a little bit odd
  - The SIMD directive tells the compiler to vectorise the code
  - It is quite different to the vectorisation hints that many compilers provide
- OpenMP `#pragma omp SIMD` is different
  - It requires the compiler to vectorise the loop
  - The vectorisation may be unsafe and wrong
  - But the compiler will make every attempt to vectorise the loop regardless of whether is it a good idea
- OpenMP is vague about exactly what it does to vectorise a loop
  - With OpenMP threads, the semantics are clear
  - The iterations of the for loop are divided between the threads

- But OpenMP is unclear about whether the loop iterations should be allocated to different lanes
  - Although that seems to be the idea of OpenMP SIMD directives

```
#pragma omp simd safelen(10)
for ( i = 10; i < n; i++ ) {
    a[i] = sqrt(a[i-10]) + b[i];
}
```

- This loop can be vectorised because the dependence distance is 10 iterations
- But it is not safe to vectorise more than 10 iterations at a time
- The "safelen" keyword tells the compiler that it should not generate vector code that operates on more than the specified number of iterations