ECE 574 – Cluster Computing
Lecture 10

Vince Weaver

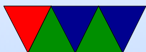
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

23 February 2017

Announcements

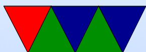
- Homework #5 was due.
- Homework #6 will be posted.



OpenMP

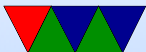
A few good references:

- https://computing.llnl.gov/tutorials/openMP/
- http://bisqwit.iki.fi/story/howto/openmp/
- http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf



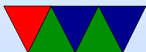
OpenMP

- Goal: parallelize serial code by just adding a few compiler directives here and there
- No need to totally re-write code like you would with pthread or MPI



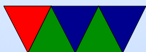
OpenMP Background

- Shared memory multi-processing interface
- C, C++ and FORTRAN
- Industry standard made by lots of companies
- OpenMP 1.0 came out in 1997 (FORTRAN) or 1998 (C), now version 4.0 (2013)
- gcc support "recently" donated, CLANG even newer
- gcc added support in 4.2 (OpenMP 2.5)
  4.4 (OpenMP 3.0), 4.7 (OpenMP 3.1), 4.9 (OpenMP 4.0), 5.0 (Offloading)



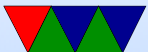
OpenMP

- Master thread with Fork/Join methodology
- Can possibly have nested threads (implementation dependent).
- Can possibly have dynamic num of threads (implementation dependent)
- Relaxed consistency, threads can cache local variables, so if you need memory to be consistent might need to flush it.



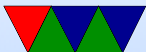
OpenMP Interface

- Compiler Directives
- Runtime Library Routines
- Environment Variables



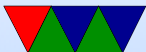
Compiler Support

- On gcc, pass `-fopenmp`
- C: `#pragma omp`
- FORTRAN: `C$OMP` or `!$OMP`



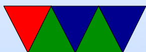
Compiler Directives

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads



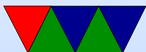
Library routines

- Need to `#include <omp.h>`
- Getting and setting the number of threads
- Getting a thread's ID
- Getting and setting threads features
- Checking in in parallel region
- Checking nested parallelism
- Locking
- Wall clock measurements



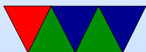
Environment Variables

- Setting number of threads
- Configuring loop iteration division
- Processor bindings
- Nested Parallelism settings
- Dynamic thread settings
- Stack size
- Wait policy

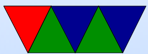


Simple Example

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>

int main (int argc, char **argv) {

        int nthreads,tid;

/* Fork a parallel region, each thread having private copy of tid */
#pragma omp parallel private(tid)
        {
        tid=omp_get_thread_num();
        printf("\tInside of thread %d\n",tid);

        if (tid==0) {
                nthreads=omp_get_num_threads();
                printf("This is the master thread, there are %d threads\n",
                        nthreads);
        }
```

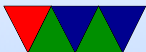


```
        }

        /* End of block, waits and joins automatically */


        return 0;
}
```



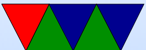
Notes on the example

- PARALLEL directive creates a set of threads and leaves the original thread the master, with tid 0.
- All threads will execute the code in parallel region
- There's an implied barrier/join at end of parallel region. Only the master continues after it.
- If any thread terminates in a parallel region, then all threads will also terminate.
- You can't `goto` into a parallel region.
- In C++ special rules on throw/catching



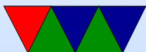
parallel directive

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)


   structured_block
```



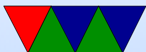
if

- if – you can do a check `if (i==0)`
  If true parallel threads are created, otherwise serially
- Why do this? Maybe it's only worth parallelizing if N greater than 16 due to overhead, can put `if (N>16)` then



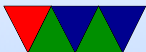
Variable Scope

- When you enter a parallel section, which variables are thread-local and which ones are globally visible?
- By default shared, but there are times you want per-thread data and not globally visible (loop indices for one)
- You specify in the parallel block how you want all of the variables to behave
  - private – variables that are private. The value is undefined at start and discarded at end

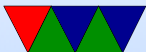


- shared – variables seen by all threads, all can be written to. Value at end is whatever the last thread wrote to it
- firstprivate – a variable inside a parallel section ends up with the value it had before the parallel section
- lastprivate – a variable after the parallel section gets the value from the last loop iteration
- copyin – you can declare special “Threadprivate” values that hold their value across parallel sections. Use this to copy the value in from the master thread.

- default – you can set to shared or none (more on C),

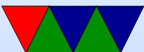


none means you have to explicitly share or private each var (makes it easier to catch bugs but more tedious)



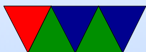
How many threads?

- Evaluation of the IF clause
- Setting of the NUM_THREADS clause
- Use of the `omp_set_num_threads()` library function
- Setting of the `OMP_NUM_THREADS` environment variable
- Implementation default – usually the number of CPUs on a node, though it could be dynamic (see next bullet).
- Threads are numbered from 0 (master thread) to N-1



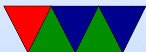
How do you actually share work?

Could do work with this, split things up manually by having a lock/critical section and divide up work per-thread. But easier way?



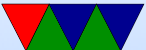
Work-sharing Constructs

- Must be inside of a parallel directive
  - do/for (do is Fortran, for is C)
  - sections
  - single – only executed by one thread
  - workshare – iterates over F90 array (Fortran90 only)



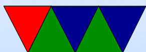
For Example

```
#include <stdio.h>
#include <stdlib.h>

#include <omp.h>

static char *memory;

int main (int argc, char **argv) {

    int num_threads=1;
    int mem_size=256*1024*1024; /* 256 MB */
    int i,tid,nthreads;

    /* Set number of threads from the command line */
    if (argc>1) {
        num_threads=atoi(argv[1]);
    }

    /* allocate memory */
    memory=malloc(mem_size);
    if (memory==NULL) perror("allocating memory");
```



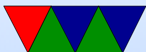
```
#pragma omp parallel shared(mem_size,memory) private(i,tid)
    {

    tid=omp_get_thread_num();
    if (tid==0) {
        nthreads=omp_get_num_threads();
        printf("Initializing %d MB of memory using %d threads\n",
            mem_size/(1024*1024),nthreads);
    }


    #pragma omp for schedule(static) nowait
    for (i=0; i < mem_size; i++)
        memory[i]=0xa5;
    }

    printf("Master thread exiting\n");

}
```

Note: loop must be simple. Integer expressions (nothing



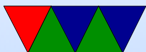
super fancy). Comparison must be only regular equals or greater/less. Iterator must be simple increment/decrement or add/subtract.

Loop iterator should be private. Why? What happens if all threads could update a global iterator?



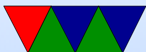
Do/For

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait

   for_loop
```

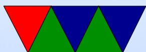


Scheduling

- By default, splits to N size/p threads chunks statically.
- schedule (static,n) chunksize n
  for example, if 10, and 100 size problem, 0-9 CPU 1, 10-19 CPU 2, 20-29 CPU3, 30-39 CPU4, 40-49 CPU1.
- But what if some finish faster than others?
- dynamic allocates chunks as threads become free. Can have much higher overhead though.
  - static – divided into size chunk, statically assigned to threads

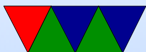


- dynamic – divided into chunks, dynamically assigned threads as they finish
- guided – like dynamic but shrinking blocksize
  why do this? When problem first starts lots of big chunks left. But near end probably not even, could end up with one thread getting large chunk and rest none. Better load balancing.
- runtime – from OMP_SCHEDULE environment variable
- auto – compiler picks for you



Other Options

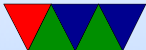
- nowait – threads do not wait at end of loop
- ordered – loops must execute in order they would in serial code
- collapse – nested loops can be collapsed
  if "perfectly nested" meaning nested with nothing inside the nests. Compiler can turn this into one big loop



Data Dependencies

Loop-carried dependencies

```
for(i=0;i<100;i++) {
    x=a[i];
    a[i+1]=x*b[i+1]; /* depends on next iteration of loop */
    a[i]=b[i];
}
```



Shift example

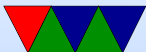
```
for(i=0;i<1000;i++)
    a[i]=a[i+1];
```

Can we parallelize this?

Equivalent, can we parallelize this?

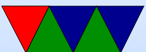
```
t[i]=a[i+1]
a[i]=t[i]
```

More overhead, but can be done in parallel



Reductions

- reduction – vector dot product. The work is split up into equal chunks, then the operator provided is used to ? and then they are all combined for final result.
  so reduction(+:a) will add up all threads as to final value

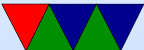


Reduction Example

```
for (int i=0;i<10;++i) {
    a = a op expr
}
```

- expr is a scalar expression that does not read a
- limited set of operations, +,-,*
- variables in list have to be shared

```
#pragma omp parallel for reduction(+:sum) schedule(static,8) num_threads(num_th$
        for(i = 0; i < N; i++) {
                /* Why does this need to be a reduction?*/
                sum = sum + i*a[i];
        }

        printf("sum=%lld\n",sum);
```

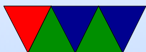


OMP Sections

You could implement this with for() and a case statement (gcc does it that way?)

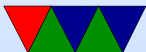
```
#pragma omp parallel sections

#pragma omp section
// WORK 1
#pragma omp section
// WORK 2
```

Will run the two sections in parallel at same time.

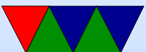


Synchronization

- OMP MASTER – only master executes instructions in this block
- OMP CRITICAL – only one thread is allowed to execute in this block
- OMP ATOMIC – like critical but for only one instruction, a memory access faster
- OMP BARRIER – force all threads to wait until all are done before continuing

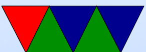


there's an implicit barrier at the end of for, section, and parallel blocks. It is useful if using nowait in loops

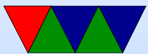


Synchronization

- Critical sections pragma omp critical (name)
- Barriers
- Locks
- `omp_init_lock()`
- `omp_destroy_lock()`
- `omp_set_lock()`

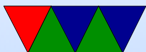


- `omp_unset_lock()`
- `omp_test_lock()`



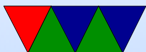
Flush directive

- `#pragma omp flush(a,b)`
- Compiler might cache variables, etc, so this forces a and b to be uptodate across threads



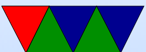
Other Notes

can call functions, functions outside of directives can still have openMP directive sin them (orphan directives)

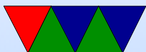


Nested Parallelism

- can have nested for loops, but by default the number of threads comes from the outer loop so an inner parallel for is effectively ignored
- can collapse loops if prefectly nested
- perfectly nested means that all computation happens in inner-most loop
- `omp_set_nested(1);` can enable nesting, but then you end up with OUTER*INNER number of threads

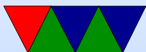


- alternately, just put the #parallel for only on the inner loop



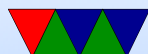
OpenMP features

- 4.0
  support for accelerators (offload to GPU, etc)
  SIMD support (specify simd)
  better error handling
  CPU affinity
  task grouping
  user-defined reductions
  sequential consistent atomics
  Fortran 2003



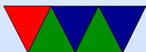
- 3.1

- 3.0
  tasks
  lots of other stuff



Pros and Cons

- Pros
  - portable
  - simple
  - can gradually add parallelism to code; serial and parallel statements (at least for loops) are more or less the same.
- Cons
  - Race conditions?



- Runs best on shared-memory systems
- Requires recent compiler