ECE 574 – Cluster Computing
Lecture 9

Vince Weaver

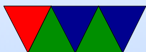
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

25 February 2021

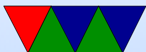
Announcements

- HW#5 will be posted, OpenMP



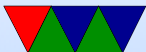
How do you actually share work?

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



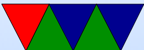
Work-sharing Constructs

- 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)



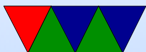
For Example

```
#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");
```



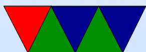
```
#pragma omp parallel shared(mem_size,memory) private(i,tid)
    {

    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



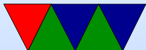
super fancy). Comparison must be only regular equals or greater/less. Iterator must be simple increment/decrement or add/subtract.

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



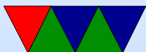
Do/For

```
#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
```

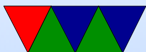


Scheduling

- 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

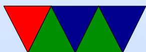


- dynamic – divided into chunks, dynamically assigned threads as they finish
- guided – like dynamic but shrinking blocksize
  why do this? When problem first starts lots of big chunks left. But near end probably not even, could end up with one thread getting large chunk and rest none. Better load balancing.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



Other Options

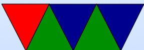
- nowait – threads do not wait at end of loop
- ordered – loops must execute in order they would in serial code
- collapse – nested loops can be collapsed
  if "perfectly nested" meaning nested with nothing inside the nests. Compiler can turn this into one big loop



Data Dependencies

Loop-carried dependencies

```
for(i=0;i<100;i++) {
    x=a[i];      /* no dependency (though careful if x is global) */
    a[i]=b[i];       /* probably no dependency but on C can alias */
    a[i]=a[i+1];     /* depends on next iteration of loop */

}
```



Shift example

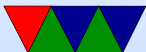
```
for(i=0;i<1000;i++)
    a[i]=a[i+1];
```

Can we parallelize this?

Equivalent, can we parallelize this?

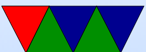
```
for(i=0;i<1000;i++)
    t[i]=a[i+1]
for(i=0;i<1000;i++)
    a[i]=t[i]
```

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

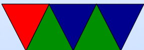


Reduction Example

```
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);
```

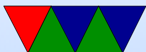


OMP Sections

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

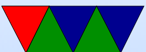
```
#pragma omp parallel sections

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

Will run the two sections in parallel at same time.

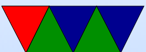


Synchronization

- 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

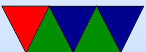


there's an implicit barrier at the end of for, section, and parallel blocks. It is useful if using nowait in loops

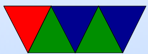


Synchronization

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

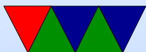


- `omp_unset_lock()`
- `omp_test_lock()`



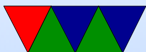
Flush directive

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



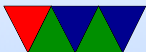
Other Notes

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

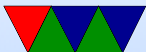


Nested Parallelism

- 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



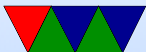
- alternately, just put the #parallel for only on the inner loop



OpenMP features

- 5.0
  task reduction
  not-equals can appear in loop comparisons

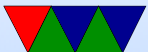
- 4.0
  support for accelerators (offload to GPU, etc)
  SIMD support (specify simd)
  better error handling
  CPU affinity
  task grouping



user-defined reductions
sequential consistent atomics
Fortran 2003

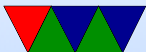
- 3.1

- 3.0
  tasks
  lots of other stuff

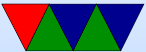


Pros and Cons

- 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?

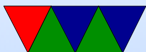


- Runs best on shared-memory systems
- Requires recent compiler



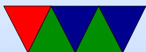
OpenMP Examples

See the course website for a link to a tarball with all the examples.



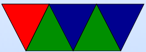
Simple

`openmp_simple.c` just creates a parallel region and prints thread number. By default, how many threasd are set up on the Haswell-EP machine?



Scope

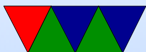
TODO: private/shared variable example



for

`openmp_for.c`

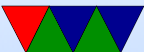
- Parallelizes the memory init loop.
- Thread number set from command line and the `num_threads()` directive.
- What happens to performance as you add threads?



static schedule

`openmp_static_schedule.c`

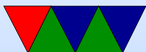
- Creates 100 threads with chunksize of 1.
- Threads are assigned loop indices at compile time.
- In example, thread 0 is fastest and 4 the slowest.
- You can see thread 0 runs through its assignment fast and then sits around doing nothing while the rest slowly finish.



dynamic schedule

`openmp_dynamic_schedule.c`

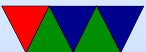
- Creates 100 threads with chunksize of 1.
- Threads are assigned loop indices dynamically.
- Each thread starts with one, but zero runs all the rest because it is so fast.



Changing Chunksize

`openmp_dynamic_chunk.c`

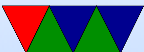
- Creates 100 threads with a prime number chunksize.
- Threads are assigned same amount of time to run.
- Spread mostly evenly but the last set of chunks, only two threads get assigned while the others have nothing to do.
- Switch to "guided" and the chunksize decreases over time and the ending is a bit more balanced.



critical

`openmp_critical.c`

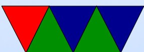
- Has a parallel loop, but a shared global counter inside.
- What happens without a critical section? (race condition)
- Put in the critical section get right results.
- But slow!
- No need to manually add mutexes, OpenMP abstracts that away.



section

```
openmp_section.c
```

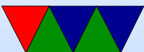
- For parallelism when you don't have a loop
- Have multiple functions that have no dependencies, want to run at same time?
- No matter how many threads you have, only can run up to the maximum number of sections at a time.



reduction

`openmp_reduction.c`

- What if you calculate something in each loop iteration, but want to sum them all in the end? Something like a vector dot product?
- You could put it in a for loop, $sum = sum + i * a[i]$ but race condition on shared sum.
- Could put in critical section but that's slow as we saw.
- Instead can use special reduction directive.

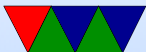


simd reduction

openmp_simd_reduction.c

https://software.intel.com/en-us/articles/enabling-simd-in-program-using-openmp40

- simd directive
- Supported by recent GCC (5.0 and later)
- Tries to map your code into SSE/AVX vector instructions if available on your processor.
- Our example turns out runs *slower*. Possibly our input set is not big enough.
- Can look at assembly code to verify it is making SIMD

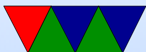


code:

`objdump --disassemble-all openmp_simd_reduction`

- Also you can use `gcc -S` to generate assembly.

  look for pmul and xmm registers



offload

Can offload to GPU or MIC.

`https://gcc.gnu.org/wiki/Offloading`

Need separate compiler for component. Support really isn't there yet.