ECE 574 – Cluster Computing
Lecture 9

Vince Weaver

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

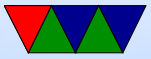
vincent.weaver@maine.edu

11am Barrows 133

13 February 2024

Announcements

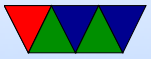
- HW#4 was posted, due Friday
- HW#2 was graded



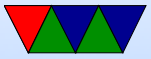
Q from last time: Returning Values from a pthread

Three common ways

- Put the result in the struct you pass to the thread, since it is visible to the main thread
- You can use the second parameter to pthread_join() which is a pointer to a void pointer (that gets tricky in C). This is returned by return from the thread, or else the argument to pthread_exit()
  - You can cast it to an int to return a single value

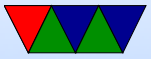


- You can malloc() a struct and return a pointer to that
- NOTE: you can't return a pointer to a local struct in the thread, as the stack will be destroyed on exit



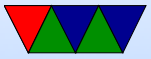
Homework #4 – Coarse Code

- Just make two threads, one for sobelx, one for sobely
- Join at end before calling combine
- This part of the homework is to make sure you understand the basics of pthread programming

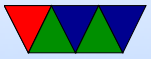


Homework #4 – Fine Code

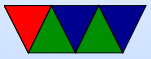
- This is much harder
- You can do this any way you want, but below is a suggested way
- If parallelizing for N threads you need to split up the work N ways
- While it might be tempting to come up with a complex scheme to have x, y, and combine all going at once, it might be best to keep it simple (and that might actually be faster)



- Easiest is to modify one of the loops to only run a subset of the values. For example, the Y loop. On thread0 run from 0..(ysize/N)-1, on thread1 from (ysize/N)..((ysize/N)*2)-1, etc
- You can modify the parameter structure so it takes a start and end value
- Have a loop that starts N threads, and calculates the start/end values above and puts them into the parameter struct before creating the thread
- Be sure for the last end value that you handle workloads not divisible by N



- You will need to have N copies of the parameter struct to use to pass to the N threads. You can't re-use one, as it's global state and you'll have a race condition. See the `pthread_join.c` example from last class for example code of this being done

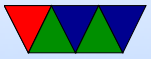


General Homework Notes – Copying Files

- For Linux/OSX easiest way to copy files from haswell-ep to your local machine is something like:

```
scp -P2131 ece574-0@weaver-lab.eece.maine.edu:hw03_submit.tar.gz .
```

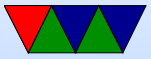
- On windows `WinSCP` is widely used
- There are various GUI SCP options for Linux/OSX too but I've never used them so can't particularly recommend any.



OpenMP

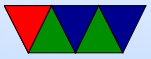
A few good references:

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



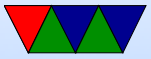
OpenMP

- 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



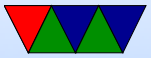
OpenMP Background

- Shared memory multi-processing interface
- Main thread with Fork/Join methodology
- C, C++ and FORTRAN
- Industry standard made by lots of companies
- OpenMP 1.0 came out in 1997 (FORTRAN) or 1998 (C), now version 5.2 (2021)



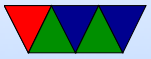
OpenMP Compiler Support

- gcc support “recently” (2008?) 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)



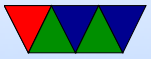
OpenMP Interface

- Compiler Directives
- Runtime Library Routines
- Environment Variables



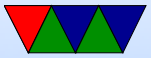
Compiler Support

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



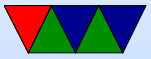
Compiler Directives

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



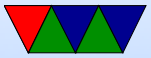
Library routines

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



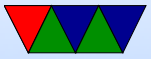
Environment Variables

- Setting number of threads (`OMP_NUM_THREADS = 4`)
- Configuring loop iteration division
- Processor bindings
- Nested Parallelism settings
- Dynamic thread settings
- Stack size
- Wait policy

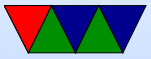


Simple Example

```
#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 main thread, there are %d threads\n",
                        nthreads);
        }
}
```

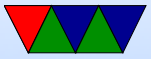


```
        /* End of block, waits and joins automatically */

        return 0;
}
```



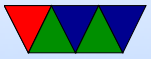
Notes on the example

- 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 main thread 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

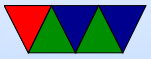
```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
    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

   structured_block
```



if

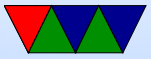
- if – you can do a check `if (i==0)`
  If true parallel threads are created, otherwise serially

- Why do this? Maybe it's only worth parallelizing if N greater than 16 due to overhead, can put `if (N>16)` then



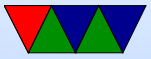
Variable Scope

- 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



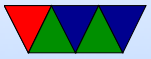
Variable Scope – private

- variables that are private
- The value is undefined at start and discarded at end
- think of it as though in the block for each thread a new variable of the same name created and all references to it are replaced by this local version



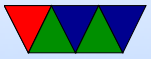
Variable Scope – shared

- variables seen by all threads
- can be written to by all threads
- Value at end is whatever the last thread wrote to it



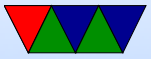
Variable Scope – firstprivate

- just like private
- the private variable inside a parallel section ends up with the value it had before the parallel section



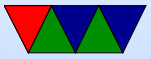
Variable Scope – lastprivate

- the variable after the parallel section gets the value from the last loop iteration



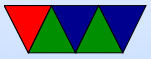
Variable Scope – copyin

- you can declare special “Threadprivate” values that hold their value across parallel sections.
- Use this to copy the value in from the master thread.



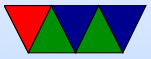
Variable Scope – default behavior

- 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)



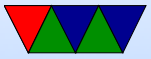
Setting number of threads

- Evaluation of the parallel `if` clause
- Setting of the parallel 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 (main thread) to N-1



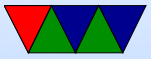
How do you actually share work?

- Could you use what we've learned to do things manually, like our pthread code?
- Using locks/critical sections and manual picking of ranges based on thread ID?
- Wouldn't it be better if the implementation could do it for us?



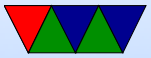
Work-sharing Constructs

- 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)

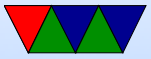


For Example

```
#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");
```

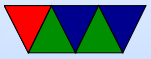


```
#pragma omp parallel shared(mem_size,memory) private(i,tid)
    {

    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");

}
```



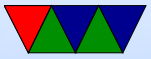
For Example Notes

- 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?



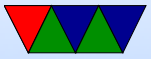
Do/For

```
#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
```



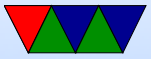
Scheduling – Static

- By default, splits up into $\frac{size}{numthreads}$ chunks statically.
- schedule (static,n) chunksize n
  - assignment of iterations to threads decided once (statically) at start of loop
  - for example, if chunksize 10, and 100 size problem: 0-9 CPU 0, 10-19 CPU 1, 20-29 CPU2, 30-39 CPU3, 40-49 CPU0.
- But what if some finish faster than others?



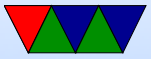
Scheduling – Dynamic

- Allocates chunks as threads become free. Can have much higher overhead though.
  - 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.



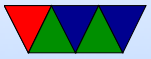
Scheduling – Other

- runtime – from OMP_SCHEDULE environment variable
- auto – compiler picks for you



Other Options for For

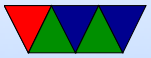
- nowait – threads do not wait at end of loop
- ordered – loops must execute in order they would in serial code
- collapse – nested loops can be collapsed
  if "perfectly nested" meaning nested with nothing inside the nests. Compiler can turn this into one big loop



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