ECE 574 – Cluster Computing
Lecture 10

Vince Weaver

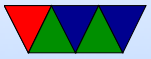
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 October 2015

Announcements

- Homework #4 will be posted eventually



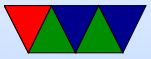
HW#4 Notes

- How granular can we be? What dependencies are there?
- How many threads should we create?
- Issues with PAPI and pthreads.
- What you need to do

  Create some threads

  Modify the convolve routines to take a struct argument

  Easiest, start each as own thread, run two at once, join, combine, done. Max speedup?
- Can also split things up. How hard can that be?

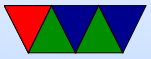


OpenMP Continued

A few good references:

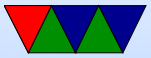
https://computing.llnl.gov/tutorials/openMP/

http://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf http://bisqwit.iki.fi/story/howto/openmp/



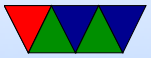
Compiler Directives

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



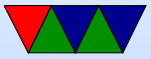
Library routines

- 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



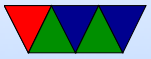
Environment Variables

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

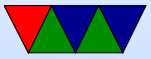


Simple Example

```
#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);
        }
```

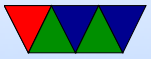


```
        }
        /* End of block, waits and joins automatically */

        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.

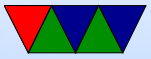


parallel clause

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

- private – variables in the list are private to each thread

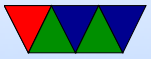


- shared – variables in the list shared across threads
  by default all are shared except loop iterator

- default – shared or none (more on C), default scope.
  none means you have to explicitly share or private each
  var

- firstprivate – a variable inside a parallel section ends up
  with the value it had before the parallel section

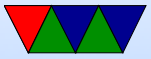
- lastprivate – a variable outside the parallel section gets
  the value from the last loop iteration



- copyin – copies in the value from the master thread into each thread (how is this different from firstprivate?)

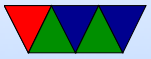
- reduction – vector dot product. The work is split up into equal chunks, then the operator provided is used to ? and then they are all combined for final result.

- num_threads – number of threads to use



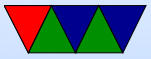
How many threads?

- 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



Work-sharing Constructs

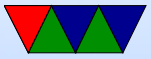
- Must be inside of a parallel directive
- do/for
- sections
- single



Do/For

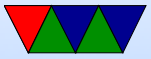
```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse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

   for_loop
```

- schedule
  static – divided into size chunk, statically assigned to threads dynamic – divided into chunks, dynamically



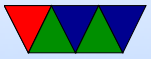
assigned threads as they finish guided – like dynamic but shrinking blocksize runtime – from OMP_SCHEDULE environment variable auto – compiler picks for you

- 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?



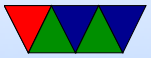
Work Constructs

- for
  reduction
- sections
- single
- master

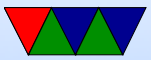


For Example

```
#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");

}
```

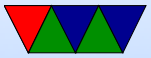


Reduction Example

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

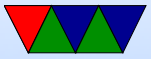
- expr is a scalar expression that does not read a
- limited set of operations, +,-,*
- variables in list have to be shared

```
#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);
```



Scheduling

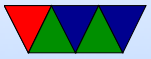
- By default, splits to N size/p threads chunks statically.
- schedule (static,n) chunksize n
  for example, if 10, and 100 size problem, 0-9 CPU 1, 10-19 CPU 2, 20-29 CPU3, 30-39 CPU4, 40-49 CPU1.
- But what if some finish faster than others?
- dynamic allocates chunks as threads become free. Can have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



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];
}
```



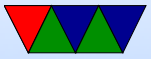
Shift example

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

Can we parallelize this?

Equivelent, can we parallelize this?

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

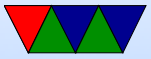


OMP Sections

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

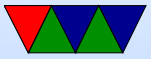
```
#pragma omp parallel sections

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

Will run the two sections in parallel at same time.

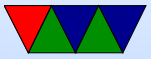


Synchronization

- 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
- OMP BARRIER – force all threads to wait until all are done before continuing

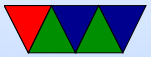


there's an implicit barrier at the end of for, section, and parallel blocks. It is useful if using nowait in loops



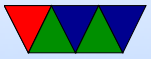
Synchronization

- Locks
- `omp_init_lock()`
- `omp_destroy_lock()`
- `omp_set_lock()`
- `omp_unset_lock()`
- `omp_test_lock()`



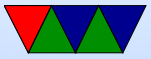
Flush directive

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

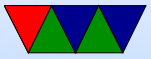


Nested Parallelism

- 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

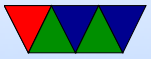


- alternately, just put the #parallel for only on the inner loop



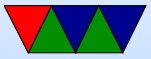
OpenMP features

- 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



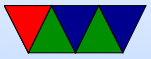
- 3.1

- 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