

ECE 574 – Cluster Computing

Lecture 9

Vince Weaver

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

vincent.weaver@maine.edu

29 September 2015

Announcements

- Homework #3 was posted.

Sorry for delay, Mainstreet down Friday night and most of Saturday



Go Over HW#3

- Convolution, good example found in Wikipedia:

https://en.wikipedia.org/wiki/Kernel_%28image_processing%29

- libjpeg and layout. Makes an array of size $x*y*3$ bytes. The three bytes (24-bits) per pixel are r,g,b
- C arrays are complex. 3D array `pixels[color][xsize][ysize]` can also (and sometimes can only) be accesses as equivalent 1D array:
`pixels[d][x][u]=pixels+((y*xsize*3)+(x*3)+d);`
- `sobelx`, `sobely`, `sobel`



- There was a minor bug in the provided sample output, was forgetting to saturate in the final step.
- Edge handling:
 - extend – the edge pixel is extended out
 - wrap – wraps around to other side of image
 - crop – output is smaller by 1 pixel border (alternately, just fill it with 0s)
 - any of these is fine, my example crops and fills with 0s
- PAPI – last level cache misses.
How many should there be?
 $320 \times 320 \times 3 \text{ bytes} = 300 \text{ kb}.$



in, outx, outy, out total = $300\text{kB} * 4 = 1.2\text{MB}$

Most likely fits in LLC (8MB on this machine, see /proc/cpuinfo).

In theory cache misses would be $1.2\text{MB} / \text{line size (64 bytes)}$

is 19200 misses in theory. Why might it be lower?

- Something cool – just make some optimization, does not need to be successful. Compare timing and L3 Total Cache Misses.



OpenMP

A good writeup is here:

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



OpenMP

- Shared memory multi-processing interface
- C, C++ and FORTRAN
- Industry standard made by lots of companies
- gcc support fairly recently donated, CLANG support even more recent
- OpenMP 1.0 came out in 1997 (FORTRAN) or 1998 (C), now version 4.0 (2013)



- 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

- Based on threads
- Master thread with Fork/Join methodology
Master thread forks off threads which compute in parallel.
When the threads complete, then join back to the master thread.
You can repeat many cycles of this.
- Include diagram



- Indicate parallelism with `#pragma omp`
- 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



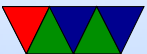
Compiler Directives

- 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

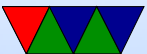


- Locking
- Wall clock measurements



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>

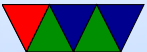
#include <omp.h>

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.



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
- 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?



For Example

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

#include <omp.h>

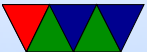
static char *memory;

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");
}

```

