ECE 574 – Cluster Computing
Lecture 7

Vince Weaver

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

vincent.weaver@maine.edu

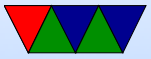
22 September 2015

Homework #2 Review

xhpl	1	2	4	8	16
real	22.4	14.8	11.4	12.9	13.0
user	21.4	22.0	24.0	50.2	50.1
sys	1.0	2.0	4.4	12.3	12.9
GFLOPs	40.6	74.9	123.8	97.4	96.4
speedup	1.0	1.5	2.0	1.7	1.7
parallel	1.0	0.75	0.5	0.21	0.1

Which row do we calculate speedup from?

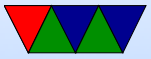
Why does it drop off at 8?



If strong scaling, then parallel efficiency would be closer to 1. Not enough results for weak scaling.

What is the worst case parallel efficiency (i.e. a single-threaded program so adding more cores does not help?). Is this truly the worst-case?

Perf record, make sure running it on benchmark, i.e. `perf record ./xhpl`
If you run on time, or the shell script you will still get the right results because perf by default follows all child processes. However if you run on sbatch, it won't work, as sbatch quickly sets things up and notifies the scheduling



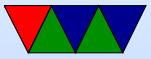
daemon via a socket connection to start things, then exits. In that case perf will only measure sbatch and not your actual benchmark run.

`perf report` will show a profiling breakdown at the function level:

```
 55.27%  xhpl     xhpl                [.] dgemm_kernel
 13.44%  xhpl     xhpl                [.] HPL_lmul
  4.38%  xhpl     [kernel.vmlinux]    [k] __lock_acquire.isra.31
  2.61%  xhpl     xhpl                [.] HPL_dlaswp00N
  2.37%  xhpl     xhpl                [.] HPL_rand
```

Pressing enter or using `perf annotate` will show at the asm level:

```
 0.00   d0:   prefet 0x200(%rdi)
```

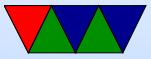


```
1.48          prefet 0x200(%r15)
1.59          prefet 0x200(%rbp)
1.60          prefet 0x200(%rsi)
0.05          vmovup (%rdi),%ymm1
0.25          vmovup 0x20(%rdi),%ymm5
0.03          vmovup (%r15),%ymm2
```

The prefetcht0 results are mysterious as usually you hope prefetches are hints and will not stall.

Let's try using a PEBS event to eliminate skid: `cycles:pp`

```
1.47 440ed0:   prefet 0x200(%rdi)
1.63 440ed7:   prefet 0x200(%r15)
1.57 440edf:   prefet 0x200(%rbp)
```

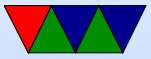


```
0.03 440ee6:   prefet 0x200(%rsi)
0.21 440eed:   vmovup (%rdi),%ymm1
```

```
440ed0:       0f 18 8f 00 02 00 00    prefetcht0 0x200(%rdi)
440ed7:       41 0f 18 8f 00 02 00    prefetcht0 0x200(%r15)
440ede:       00
440edf:       0f 18 8d 00 02 00 00    prefetcht0 0x200(%rbp)
440ee6:       0f 0d 8e 00 02 00 00    prefetchw 0x200(%rsi)
440eed:       c5 fc 10 0f             vmovups (%rdi),%ymm1
440ef1:       c5 fc 10 6f 20          vmovups 0x20(%rdi),%ymm5
```

prefetcht0 – fetches into all levels of cache

Even using a prime number for the sample frequency

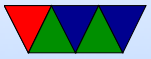


didn't help. (4481, default is 4000).

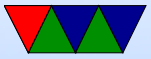
Looking at `instructions:pp` gives more interesting results, but this is instruction frequency so ignores stall latencies.

```
 68.80%  xhpl     xhpl                  [.] dgemm_kernel
 14.09%  xhpl     xhpl                  [.] HPL_lmul
  3.59%  xhpl     xhpl                  [.] HPL_rand
  1.74%  xhpl     xhpl                  [.] HPL_ladd
  1.46%  xhpl     [kernel.vmlinux]      [k] __lock_acquire.
```

```
 0.65 44160c:   vfmadd %ymm0,%ymm1,%ymm5
 0.29 441611:   vfmadd %ymm0,%ymm2,%ymm9
 0.66 441616:   vfmadd %ymm0,%ymm3,%ymm13
```



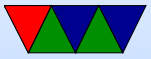
```
0.11 44161b:    vpermp $0x1b,%ymm0,%ymm0
1.15 441621:    vfmadd %ymm0,%ymm1,%ymm6
0.31 441626:    vfmadd %ymm0,%ymm2,%ymm10
0.22 44162b:    add    $0x40,%rdi
0.65 44162f:    vfmadd %ymm0,%ymm3,%ymm14
```



Pthread Programming

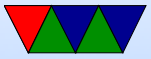
- based on this really good tutorial here:

  `https://computing.llnl.gov/tutorials/pthreads/`



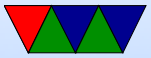
Pthread Programming

- Changes to shared system resources affect all threads in a process (such as closing a file)
- Identical pointers point to same data
- Reading and writing to same memory is possible simultaneously (with unknown origin) so locking must be used



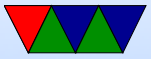
When can you use?

- Work on data that can be split among multiple tasks
- Work that blocks on I/O
- Work that has to handle asynchronous events



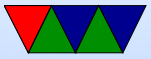
Models

- Pipeline – task broken into a set of subtasks that each execute serial on own thread
- Manager/worker – a manager thread assigns work to a set of worker threads. Also manager usually handles I/O static worker pool – constant number of threads dynamic worker pool – threads started and stopped as needed
- Peer – like manager/worker but the manager also does calculations



Shared Memory Model

- All threads have access to shared memory
- Threads also have private data
- Programmers must properly protect shared data



Thread Safeness

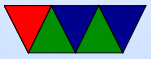
Is a function called thread safe?

Can the code be executed multiple times simultaneously?

The main problem is if there is global state that must be remembered between calls. For example, the strtok() function.

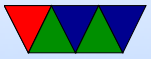
As long as only local variables (on stack) usually not an issue.

Can be addressed with locking.

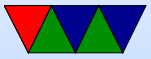


POSIX Threads

- 1995 standard
- Various interfaces:
  1. Thread management: Routines for manipulating threads – creating, detaching, joining, etc. Also for setting thread attributes.
  2. Mutexes: (mutual exclusion) – Routines for creating mutex locks.
  3. Condition variables – allow having threads wait on a lock

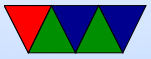


4. Synchronization: lock and barrier management



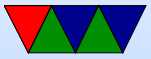
POSIX Threads (pthreads)

- A C interface. There are wrappers for Fortran.
- Over 100 functions, all starting with pthread_
- Involve "opaque" data structures that are passed around.
- Include pthread.h header
- Include -pthread in linker command to compiler



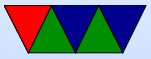
Creating Threads

- Your function, as per normal, only includes one thread
- `pthread_create()` creates a new thread
- You can call it anywhere, as many times as you want
- pthread_create (thread,attr,start_routine,arg)
- You pass is a pointer to a thread object (which is opaque), an attr object (which can be NULL), a



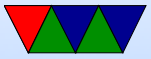
start_routine which is a C function called when it starts, an an arg argument to pass to the routine.

- Only can pass one argument. How can you pass more? pointer to a structure.
- With attributes you can set things like scheduling policies
- No routines for binding threads to specific cores, but some implementations include optional non-portable way. Also Linux has sched_setaffinity routine.



Terminating Threads

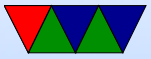
- pthread_exit()
- Returns normally from its starting routine
- another thread uses pthread_cancel() in it
- The entire process is terminated (by ending, or calling exit(), etc)



Thread Management

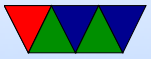
- pthread_join() lets a thread block until another one finishes

  So master can join all the children and wait until they are done before continuing.



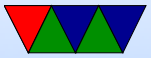
Stack Management

- Manage your own stack? Can get and set size. Be careful allocating too much on stack.



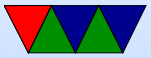
Mutexes

- Type of lock, only one thread can own it at a time. Can be used to avoid race conditions.



Condition Variables

- A way to avoid spinning on a mutex



Debugging