

# ECE 574 – Cluster Computing

## Lecture 7

Vince Weaver

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

`vincent.weaver@maine.edu`

12 February 2019

# Announcements

- Homework #3 was posted. Don't put it off until the last minute!



# Homework #2 Review

● 2

1.

Procs	1	2	3	4	8	16	32	64
Time	1:58	1:03	0:42	0:34	0:21	0:14	0:16	0:16
Time	118	63	42	34	21	14	16	16
GFLOPS	45	84	125	157	260	368	336	325
Speedup	—	1.87	2.81	3.47	5.62	8.42	7.37	7.37
Peff	—	0.94	0.94	0.87	0.70	0.53	0.23	0.11

2. 2b) Speedup:  $(t_1/t_p)$

3. 2c) Parallel effic:  $(S_p/p$  or  $T_1/pT_p)$

4. 2d) Yes, time decreases as you add cores.

Not ideal strong scaling though.



5. 2e) No weak, didn't test with sizes constant
  6. Time is less as only dgemm, not malloc or randomizing
  7. More because user adds all cores
- 3
    - 3a) dgemm kernel (double-precision generic matrix-matrix multiply. algorithm kernel (core) not Linux kernel)  
If you got bit time in kernel, you ran perf on time
    - 3b) vmovups (truncated) (memcpy?)
    - 3c) skid



# Homework #2 More

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

To get 1G/core, roughly  $\frac{2}{3} * n^3 = 500B * p$

Cores		N=20k	Size=3.2G	Size=1G/core	time	Speedup	GFLOPs
1	119s	3.2G	11,000	9000	11.5	—	42.5
2	64s	1.6G	16,000	11500	14	0.82	72
4	37s	0.8G	22,360	14400	14	0.82	139
8	22s	0.4G	31,600	18200	18	0.63	222
16	18s	0.2G	44,700	22900	26	0.44	304
32	18s	0.1G	63,240	28800	47	0.24	336
64	18s	0.05G	89,000	36000	93	0.12	334



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:

Samples: 688K of event 'cycles', Event count (approx.): 565000306359

Overhead	Command	Shared Object	Symbol
81.14%	xhpl	xhpl	[.] dgemm_kernel
3.39%	xhpl	xhpl	[.] HPL_lmul
3.09%	xhpl	xhpl	[.] HPL_rand
1.57%	xhpl	xhpl	[.] HPL_dlaswp00N
1.53%	xhpl	xhpl	[.] HPL_ladd
1.48%	xhpl	libc-2.28.so	[.] __sched_yield
1.16%	xhpl	xhpl	[.] HPL_setran
0.66%	xhpl	xhpl	[.] HPL_pdlange
0.60%	xhpl	[unknown]	[k] 0xfffffe0000006000

Pressing enter or using perf annotate will show at the



## asm level:

```
0.03 | c0:   vmovups (%rdi),%ymm1
0.62 |      vmovups 0x20(%rdi),%ymm2
0.07 |      vmovups (%r15),%ymm3
0.93 |      vmovups %ymm1,(%rsi)
0.03 |      vmovups %ymm2,0x20(%rsi)
0.02 |      vmovups %ymm3,0x40(%rsi)
0.07 |      add     $0x40,%rdi
0.00 |      add     $0x40,%r15
0.02 |      add     $0x60,%rsi
0.01 |      dec     %rax
      |      jne     c0
```

move unaligned packed 256-bits from memory to register (single precision?)

memory copy? By default uses :ppp to reduce skid





# Parallel Programming!



# Processes – a Review

- Multiprogramming – multiple processes run at once
- Process has one view of memory, one program counter, one set of registers, one stack
- Context switch – each process has own program counter saved and restored as well as other state (registers)
- OSes often have many things running, often in background.



On Linux/UNIX sometimes called daemons  
Can use top or ps to view them.

- Creating new: on Unix its fork/exec, windows CreateProcess
- Children live in different address space, even though it is a copy of parent
- Process termination: what happens?  
Resources cleaned up. atexit routines run.  
How does it happen?



`exit()` syscall (or return from main).

Killed by a signal.

Error

- Unix process hierarchy.

Parents can wait for children to finish, find out what happened

not strictly possible to give your children away, although `init` inherits orphans

- Process control block.



# Threads

- Each process has one address space and single thread of control.
- It might be useful to have multiple threads share one address space
  - GUI: interface thread and worker thread?
  - Game: music thread, AI thread, display thread?
  - Webserver: can handle incoming connections then pass serving to worker threads
  - Why not just have one process that periodically switches?



- Lightweight Process, multithreading
- Implementation:  
Each has its own PC  
Each has its own stack
- Why do it?  
shared variables, faster communication  
multiprocessors?  
mostly if does I/O that blocks, rest of threads can keep going  
allows overlapping compute and I/O



- Problems:

What if both wait on same resource (both do a scanf from the keyboard?)

On fork, do all threads get copied?

What if thread closes file while another reading it?



# Thread Implementations

- Cause of many flamewars over the years





# User-Level Threads (N:1 one process many threads)

- Benefits

- Kernel knows nothing about them. Can be implemented even if kernel has no support.
- Each process has a thread table
- When it sees it will block, it switches threads/PC in user space
- Different from processes? When `thread_yield()` called it can switch without calling into the kernel (no slow



kernel context switch)

- Can have own custom scheduling algorithm
- Scale better, do not cause kernel structures to grow

- Downsides

- How to handle blocking? Can wrap things, but not easy. Also can't wrap a pagefault.
- Co-operative, threads won't stop unless voluntarily give up.

Can request periodic signal, but too high a rate is inefficient.



# Kernel-Level Threads (1:1 process to thread)

- Benefits
  - Kernel tracks all threads in system
  - Handle blocking better
- Downsides
  - Thread control functions are syscalls
  - When yielding, might yield to another process rather than a thread



– Might be slower



# Hybrid (M:N)

- Can have kernel threads with user on top of it.
- Fast context switching, but can have odd problems like priority inversion.



# Linux

- Posix Threads
- Originally used only userspace implementations. GNU portable threads.
- LinuxThreads – use clone syscall, SIGUSR1 SIGUSR2 for communicating.  
Could not implement full POSIX threads, especially with signals. Replaced by NPTL  
Hard thread-local storage



Needed extra helper thread to handle signals

Problems, what happens if helper thread killed? Signals broken? 8192 thread limit? proc/top clutter up with processed, not clear they are subthreads

- NPTL – New POSIX Thread Library

Kernel threads

Clone. Add new futex system calls. Drepper and Molnar at RedHat

Why kernel? Linux has very fast context switch compared to some OSes.

Need new C library/ABI to handle location of thread-



local storage

On x86 the fs/gs segment used. Others need spare register.

Signal handling in kernel

Clone handles setting TID (thread ID)

exit\_group() syscall added that ends all threads in process, exit() just ends thread.

exec() kills all threads before execing

Only main thread gets entry in proc





# 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.
- Argument to a join is a specific thread to wait on (so if waiting on four, have to have four calls to `pthread_join()`)



# 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



# Race Conditions

- Shared counter address  
RMW on ARM  
Thread A reads value into reg  
Context switch happens  
Thread B reads value into reg, increments, writes out  
Context switch back to A  
increments value, writes out  
What happened?  
What should value be?



# Critical Sections

- Want mutual exclusion, only one can access structure at once
  1. no two processes can be inside critical section at once
  2. no assumption can be made about speed of CPU
  3. no process not in critical section may block other processes
  4. no process should wait forever



# How to avoid

- Disable interrupts. Heavy handed, only works on single-core machines.
- Locks/mutex/semaphore



# Mutex

- `mutex_lock`: if unlocked (0), then it sets lock and returns  
if locked, returns 1, does not enter.  
what do we do if locked? Busy wait? (spinlock) re-  
schedule (yield)?
- `mutex_unlock`: sets variable to zero



# Semaphore

- Up/Down
- Wait in queue
- Blocking
- As lock frees, the job waiting is woken up



# Locking Primitives

- fetch and add (bus lock for multiple cores), xadd (x86)
- test and set (atomically test value and set to 1)
- test and test and set
- compare-and-swap – Atomic swap instruction SWP  
(ARM before v6, deprecated)  
x86 CMPXCHG  
Does both load and store in one instruction!



Why bad? Longer interrupt latency (can't interrupt atomic op)

Especially bad in multi-core

- load-link/store conditional

Load a value from memory

Later store instruction to same memory address. Only succeeds if no other stores to that memory location in interim.

ldrex/strex (ARMv6 and later)

- Transactional Memory





# Locking Primitives

- can be shown to be equivalent
- how swap works:  
lock is 0 (free).  $r1=1$ ; swap  $r1,lock$   
now  $r1=0$  (was free),  $lock=1$  (in use)  
lock is 1 (not-free).  $r1=1$ , swap  $r1,lock$   
now  $r1=1$  (not-free),  $lock$  still==1 (in use)



# Memory Barriers

- Not a lock, but might be needed when doing locking
- Modern out-of-order processors can execute loads or stores out-of-order
- What happens a load or store bypasses a lock instruction?
- Processor Memory Ordering Models, not fun
- Technically on BCM2835 we need a memory barrier any time we switch between I/O blocks (i.e. from serial



to GPIO, etc.) according to documentation, otherwise loads could return out of order



# Deadlock

- Two processes both waiting for the other to finish, get stuck
- One possibility is a bad combination of locks, program gets stuck
- P1 takes Lock A. P2 takes Lock B. P1 then tries to take lock B and P2 tries to take Lock A.



# Livelock

- Processes change state, but still no forward progress.
- Two people trying to avoid each other in a hall.
- Can be harder to detect



# Starvation

- Not really a deadlock, but if there's a minor amount of unfairness in the locking mechanism one process might get "starved" (i.e. never get a chance to run) even though the other processes are properly taking and freeing the locks.



# How to avoid Deadlock

- Don't write buggy code
- Pre-emption (let one of the stuck processes run anyway)
- Rollback (checkpoint occasionally)
- What to do if it happens?
  - Reboot the system
  - Kill off stuck processes

