

# ECE 574 – Cluster Computing

## Lecture 7

Vince Weaver

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

`vincent.weaver@maine.edu`

18 February 2025

# Announcements

- Class on Thursday was cancelled due to snowstorm
- Homework #3 was extended to Monday (yesterday)
- Lots of coding
- Later homeworks will build off of it, but don't worry I will provide solutions



# Homework #3 Notes

- You can output intermediate sobel results for debugging
- If almost right except for a few areas, most likely forgot to saturate  
note you need to saturate after combine as well  
saturate to 255
- If you run into trouble, send me your code to look at
- PAPI: be sure to do things in right order
- PAPI: if get weird results, be sure to check error returns from the functions. There's a `PAPI_sterror()` that



can help



# Homework #3 Notes

- Sample image, Butterfinger who was a guinea pig I had as a pet many years ago
- There are some “standard” sample images. Most famous is the Lena image which has a complicated backstory.



# Homework #2 Review – Measurements

Procs	1	2	4	8	16	32	64
Time	115	78	34	20	14	16	15
GFLOPS	46	70	159	268	373	341	351
Speedup	—	1.5	3.5	5.8	8.1	7.4	7.6
Peff	—	0.75	0.87	0.73	0.50	0.23	0.12

- 3bi) Speedup:  $(t_1/t_p)$

This year moving to 32 threads didn't help

What is different? Kernel? Compiler? OpenBLAS?

Firmware?

- 3bii) Parallel efficiency:  $(Sp/p$  or  $T_1/pT_p)$
- 3biii) Yes, time decreases as you add cores.



Not ideal strong scaling though.

Note: “weak” is not the same as “poor strong scaling”

- 3biv) No weak, didn't test with sizes constant

How could we test this?

- 3bv Time is less as only dgemm, not malloc or randomizing
- 3bvi More because user adds up all threads/cores



# Homework #2 Review – perf record

- 3a) dgemm kernel (double-precision generic matrix-matrix multiply. algorithm kernel (core) not Linux kernel)  
If you got big time in kernel, you ran perf on time

```
59.96% xhpl      xhpl      [.] dgemm_kernel
19.41% xhpl      xhpl      [.] inner_thread
 6.67% xhpl      xhpl      [.] blas_thread_server
 3.83% xhpl      xhpl      [.] HPL_lmul
 1.99% xhpl      xhpl      [.] exec_blas_async_wait
 1.44% xhpl      xhpl      [.] HPL_rand
 1.34% xhpl      xhpl      [.] HPL_dlaswp00N
 1.15% xhpl      xhpl      [.] HPL_ladd
 0.93% xhpl      xhpl      [.] HPL_setran
 0.52% xhpl      [kernel.kallsyms] [k] clear_page_erms
```





# Homework #2 Review – perf annotate

```
0.24 -          vfmadd231pd  %ymm0,%ymm1,%ymm4
0.27 -          vfmadd231pd  %ymm0,%ymm2,%ymm8
0.29 -          vfmadd231pd  %ymm0,%ymm3,%ymm12
0.22 -          vbroadcastsd -0x58(%rdi),%ymm0
0.44 -          vfmadd231pd  %ymm0,%ymm1,%ymm5
0.20 -          vfmadd231pd  %ymm0,%ymm2,%ymm9
0.35 -          vfmadd231pd  %ymm0,%ymm3,%ymm13
0.09 -          vbroadcastsd -0x50(%rdi),%ymm0
0.35 -          vfmadd231pd  %ymm0,%ymm1,%ymm6
```

- in `dgemm_kernel()`  
vbroadcastsd – broadcast 64-bit fp value from memory  
and copy 4 times in 256-bit AVX register



vfmadd231pd – fused multiply-add of packed doubles.  
231 refers to the order of the operands ( $2*3+1$ , store in 1)

- 3c) skid

- Will :pp avoid the skid?

- Why no one thing stand out in profile?

Hand optimized assembly, people have worked a long time on optimizing this getting low hanging stuff

- Both instructions only listed as latency 1 in the agner fogg, though that doesn't count cache access



# Homework #2 – Weak Scaling Info

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



# From Last time

- Go over SMT briefly
- Go over NUMA briefly
- Go over Shared Memory vs Distributed Briefly



# Parallel Programming!



# Some brief Operating System Review

- The operating system shields us from a lot of the low-level hardware issues with parallel programming
- Likewise we'll use libraries to hide a lot of the operating-system level stuff going on
- I'll still give some background because it can be helpful to know, if you're really interested you can take ECE531 Advanced Operating Systems where we learn about some of this in detail



# Single-Thread Processes

- A process is a program running on a computer, usually being managed by an operating system
- Process has one view of memory, one program counter, one set of registers, one stack

TODO: diagram?



# Multi-tasking / Multi-Programming

- Most OSes give illusion of running multiple processes at once (even on a single core system)
- Rapidly switch between all running processes, hundreds of times a second
- Context switch – each process has own program counter saved and restored as well as other state (registers)
- Virtual Memory is used to give each process illusion they have sole access to memory in the machine
- OSes often have many things running, often in





background.

On Linux/UNIX sometimes called daemons

Can use `top` or `ps` to view them.



# Processes: OS Interface

- Creating new: on Unix `fork()/exec()`, Windows `CreateProcess`
- Child processes live in different address space, even though it is a copy of parent
- Process termination: what happens?
  - Resources cleaned up. `atexit()` routines run.
  - `exit()` syscall (or return from main).
  - Can also be killed by a signal or 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



# Could you build a multi-cpu program using just Processes?

- Yes. Start or fork() many copies, and have them communicate via message passing (this is more like a distributed system)
- Use Inter-Process Communication (IPC) Linux has many, all have tradeoffs
  - Sockets (UNIX domain, net)
  - Anonymous memory (mmap)
  - Files (on disk or mmaped)



- Signals
- Pipes (Anonymous, Named, FIFOs)
- Shared Memory (SysV, POSIX)
- Message Queues (SysV, POSIX)
- Semaphores (SysV, POSIX)
- Futex locks
- Inotify
- FUSE
- D-Bus
- others?



# Threads

- Can an address space have multiple threads of control running in it at once?
- Examples when this might be useful:
  - GUI: interface thread and worker thread?
  - Game: music thread, AI thread, display thread?
  - Webserver: can handle incoming connections then pass serving to worker threads



# Multithreading Implementation

- The memory layout is shared by all threads in a process
- Each thread has its own PC
- Each thread has its own stack
- Each thread has its own copy of the register file
- Each thread has its own “thread local storage” (TLS) area for private variables



# Multithreading Tradeoffs

- Benefits
  - shared variables, faster communication
  - If program blocks on I/O, rest of threads can keep going
  - programs can run faster on multiprocessor systems
- Complications
  - Resource conflicts: (what if multiple threads try to `scanf()` at same time?)
  - What if a thread closes a file while another is trying





to read?

- On a fork, does the child process have the same threads or not?



# 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.



# Common Thread Programming 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 Safety

- 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 functions only use local variables (on stack, not static or global) usually not an issue.
- Some issues can be addressed with locking.



# POSIX Threads (pthreads)

- Standardized thread interface
- Standard cross-platform set of routines to use



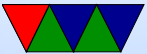
# Other types of Threads

- co-routines
  - Not found in C
  - Sort of like co-operatively scheduled software threads
  - functions can be suspended at various points and re-started later
  - Less issues than full threads as only one can run at a time, simplifying locking
- C11/C17 threads
  - <https://beej.us/guide/bgc/html/split/multithread>



html

- Sort of last-minute addition to C
- No real benefits over using pthreads?
- Fibers?



# Linux Threading – Historical

- Linux original thread implementation was horrible software based
- 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  
processes, not clear they are subthreads



# Linux Threading – NPTL

- NPTL – Native POSIX Thread Library
- Kernel threads
- Clone syscall, new futex system calls.
- Developed around 2003 or so by Drepper and Molnar at RedHat, Kernel 2.6
- 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

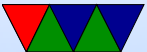




# Pthread Programming

- based on this really good tutorial here:

<https://hpc-tutorials.llnl.gov/posix/>



# Pthread Programming Challenges

- 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



# POSIX Threads (1995)

- 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

