ECE 574 – Cluster Computing Lecture 6

Vince Weaver

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

`vincent.weaver@maine.edu`

16 February 2021

Announcements

- Homework #3 was posted. Don't put it off until the last minute!
- Lots of coding
- Later homeworks will build off of it, but don't worry I will procvide solutions

Homework #2 Review

- 2
  1.

Procs	1	2	3	4	8	16	32	64
Time	115	61	??	34	22	18	18	??
GFLOPS	46	88	???	159	244	299	297	???
Speedup	—	1.9	???	3.4	5.2	6.4	6.4	????
Peff	—	0.95	???	0.85	0.65	0.40	0.20	???

  2. 2b) Speedup: (t1/tp)
  3. 2c) Parallel effic: (Sp/p or T1/pTp)
  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 up all threads/cores

- 3
  - 3a) dgemm kerenel (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)

```
0.30 |        vbroadcastsd -0x60(%rdi),%ymm0
0.23 |        vfmadd231pd  %ymm0,%ymm1,%ymm4
0.27 |        vfmadd231pd  %ymm0,%ymm2,%ymm8
0.28 |        vfmadd231pd  %ymm0,%ymm3,%ymm12
0.22 |        vbroadcastsd -0x58(%rdi),%ymm0
0.43 |        vfmadd231pd  %ymm0,%ymm1,%ymm5
```

```
  0.20 |        vfmadd231pd  %ymm0,%ymm2,%ymm9
  0.36 |        vfmadd231pd  %ymm0,%ymm3,%ymm13
  0.09 |        vbroadcastsd -0x50(%rdi),%ymm0
```

in dgemm_kernel() vbroadcastd – broardcast fp value in memory 4 times in register vfmadd231pd – fused multiply-add of packed doubles. 231 refers to the order of the operands

- ◦ 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

Cache Coherency

- How do you handle data being worked on by multiple processors, each with own cache of main memory?
- Cache coherency protocols.
- Many and varied. MESI is a common one
- Directory vs Snoopy

MESI

- Modified, Exclusive, Shared, Invalid

Barriers and Ordering

- On modern out-of-order execution, memory accesses can happen out-of-order
- Sequential consistency – all happen in order
- Strong consistency – stores
- Weak consistency – can be arbitrarily reordered, only barriers protect you
- A memory barrier instruction makes sure all previous

loads/stores finish before moving on

- Most important for things like locks, as well as memory-mapped I/O

Ordering Example

```
    y1=0
    y2=0
    y1=3
    y2=4
Another core
    x1=y1
    x2=y2
```

What values of x1 and x2 can you get?

Strong:

x1=0,x2=0

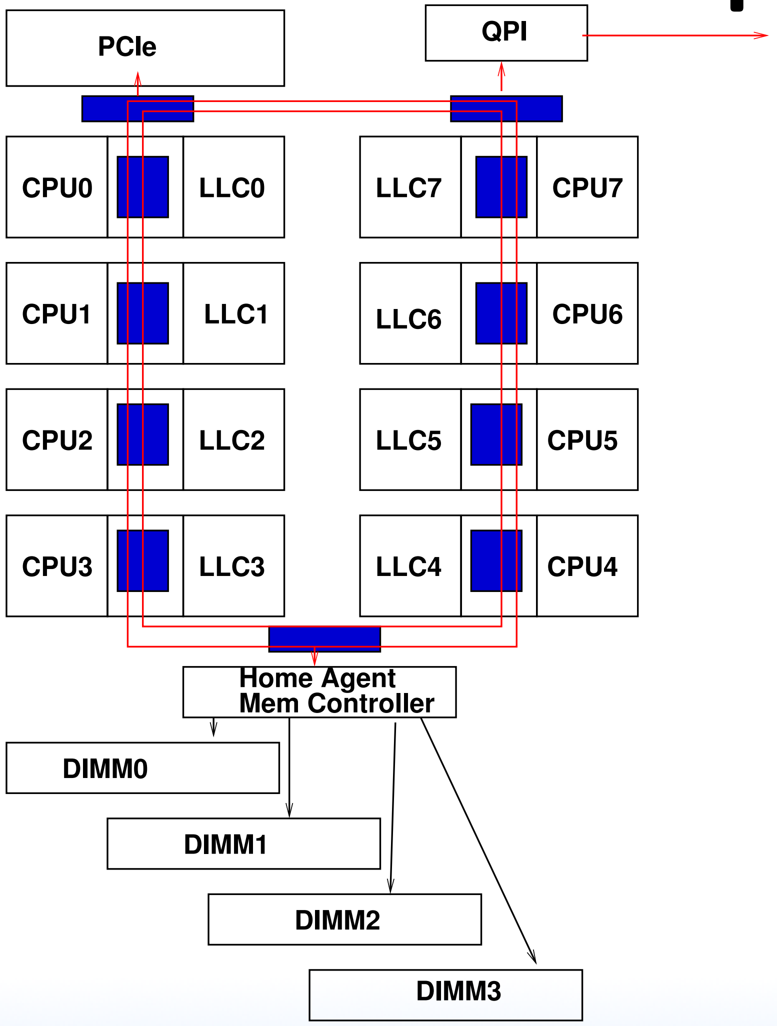
x1=3,x2=0

x1=3,x2=4

Weak:

x1=0,x2=4

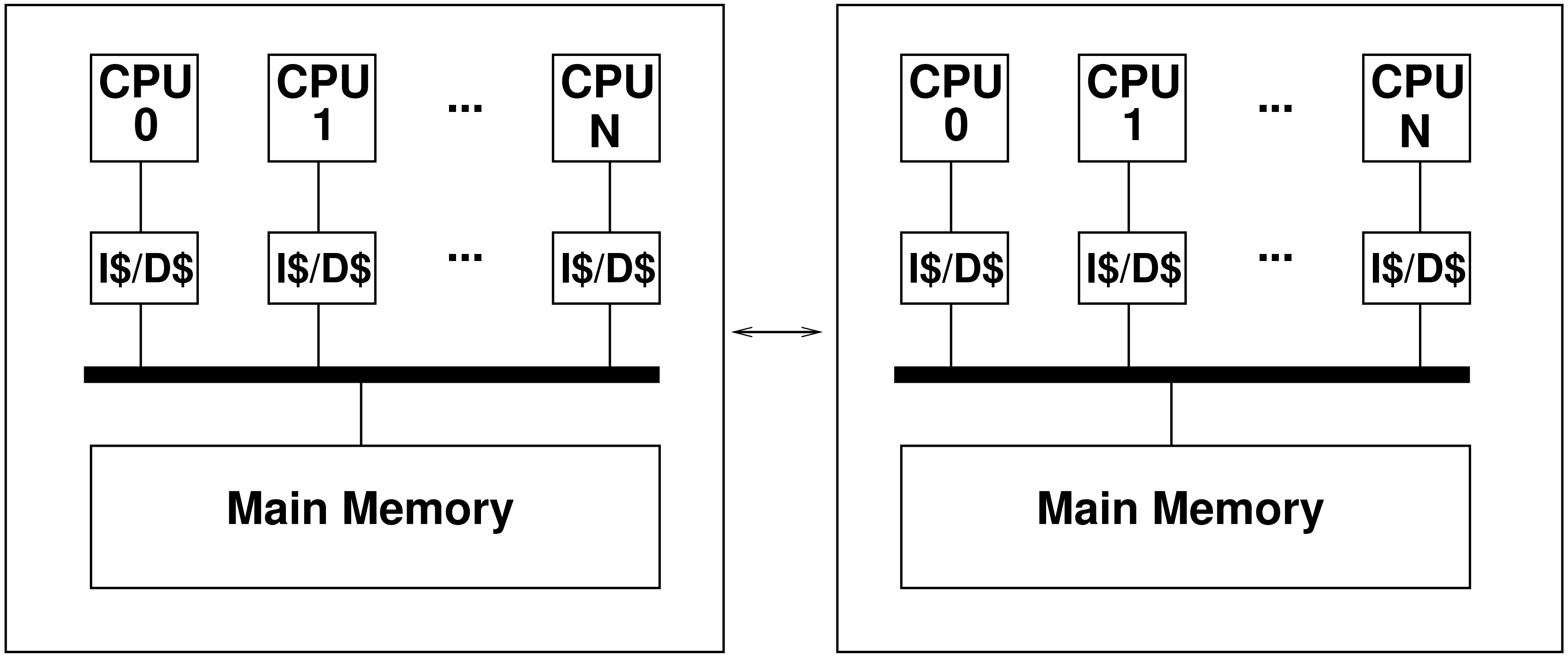
Haswell EP Setup



NUMA

Non-uniform memory access – some accesses will have to cross to other processors, causing extra delay. How can you optimize this?

Traditional NUMA Layout



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