CS 61C: Great Ideas in Computer Architecture

OpenMP, Transistors

Instructor: Justin Hsia

Review of Last Lecture

- Amdahl's Law limits benefits of parallelization
- Multiprocessor systems uses shared memory (single address space)
- Cache coherence implements shared memory even with multiple copies in multiple caches
  - Track state of blocks relative to other caches (e.g. MOESI protocol)
  - False sharing a concern
- Synchronization via hardware primitives:
  - MIPS does it with Load Linked + Store Conditional

Question: Consider the following code when executed *concurrently* by two threads.

What possible values can result in *($s0)?

```
# *($s0) = 100
lw   $t0,0($s0)
addi $t0,$t0,1
sw   $t0,0($s0)
```

- ☐ **101 or 102**
- ☐ **100, 101, or 102**
- ☐ **100 or 101**
- ☐ **102**

Great Idea #4: Parallelism

Software

- Parallel Requests
  - Assigned to computer
  - e.g. search "Katz"
- Parallel Threads
  - Assigned to core
  - e.g. lookup, ads
- Parallel Instructions
  - > 1 instruction @ one time
  - e.g. 5 pipelined instructions
- Parallel Data
  - > 1 data item @ one time
  - e.g. add of 4 pairs of words
- Hardware descriptions
  - All gates functioning in parallel at same time

Leverage Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Smart Phone

Computer

Core … Core

Memory

Input/Output

Core

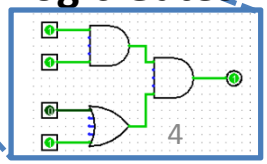
Instruction Unit(s)

Functional Unit(s)

A_0+B_0 | A_1+B_1 | A_2+B_2 | A_3+B_3

Cache Memory

Logic Gates

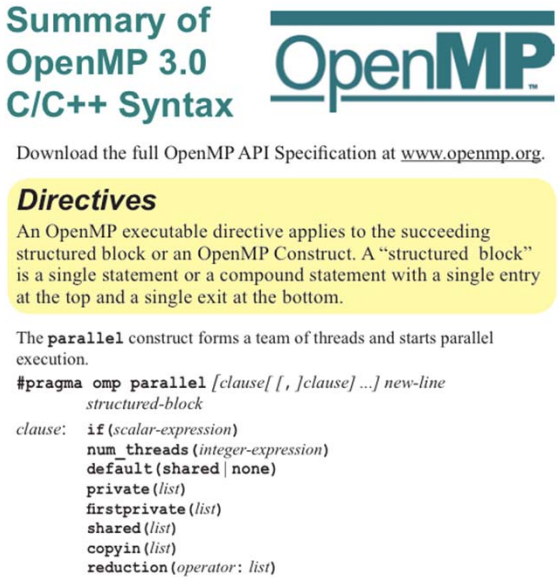


Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Directives
  - Workshare
  - Synchronization
- OpenMP Common Pitfalls
- Hardware: Transistors
- Bonus: `sections` Example

What is OpenMP?

- API used for multi-threaded, shared memory parallelism
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable
- Standardized
- **Resources:** http://www.openmp.org/ and http://computing.llnl.gov/tutorials/openMP/



Summary of OpenMP 3.0 C/C++ Syntax

OpenMP

Download the full OpenMP API Specification at www.openmp.org.

Directives

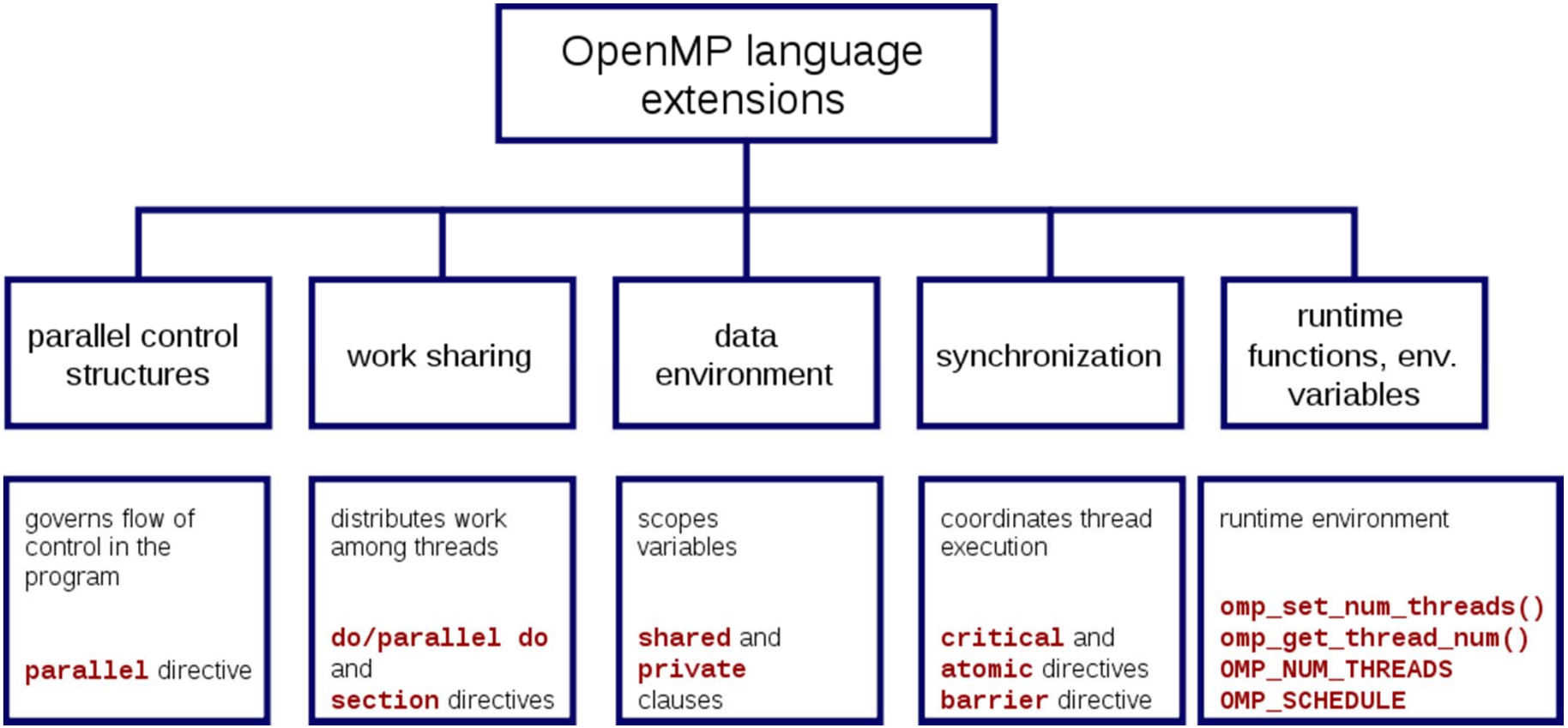
An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A "structured block" is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

The **parallel** construct forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[ [ , ]clause] ...] new-line
        structured-block
```

clause:
```
if (scalar-expression)
num_threads (integer-expression)
default (shared | none)
private (list)
firstprivate (list)
shared (list)
copyin (list)
reduction (operator: list)
```

OpenMP Specification



Shared Memory Model with Explicit Thread-based Parallelism

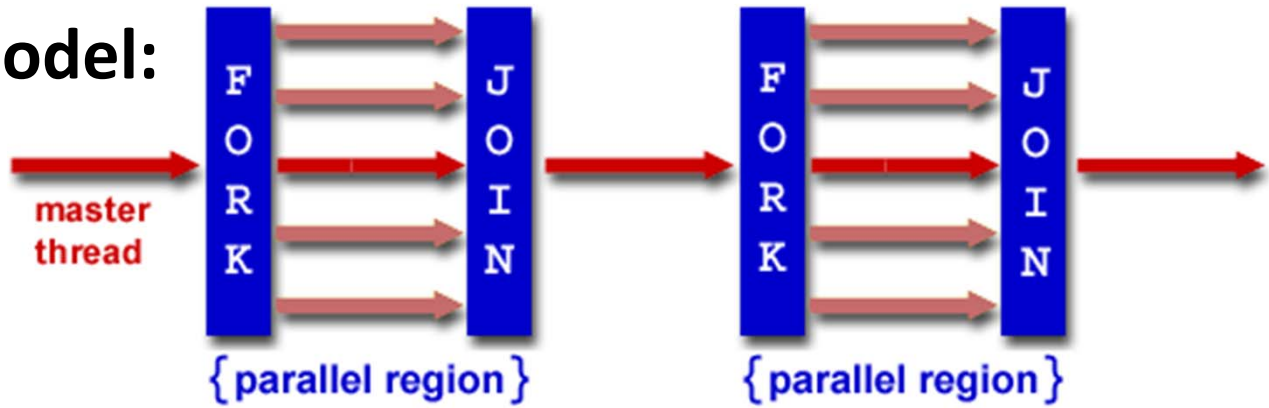
- Multiple threads in a shared memory environment, explicit programming model with full programmer control over parallelization
- **Pros:**
  - Takes advantage of shared memory, programmer need not worry (that much) about data placement
  - Compiler directives are simple and easy to use
  - Legacy serial code does not need to be rewritten
- **Cons:**
  - Code can only be run in shared memory environments
  - Compiler must support OpenMP (e.g. gcc 4.2)

OpenMP in CS61C

- OpenMP is built on top of C, so you don't have to learn a whole new programming language
  - Make sure to add `#include <omp.h>`
  - Compile with flag: `gcc -fopenmp`
  - Mostly just a few lines of code to learn
- You will NOT become experts at OpenMP
  - Use slides as reference, will learn to use in lab
- **Key ideas:**
  - Shared vs. Private variables
  - OpenMP directives for parallelization, work sharing, synchronization

OpenMP Programming Model

- **Fork - Join Model:**



- OpenMP programs begin as single process (*master thread*) and executes sequentially until the first parallel region construct is encountered
  - *FORK:* Master thread then creates a team of parallel threads
  - Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads
  - *JOIN:* When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

OpenMP Extends C with Pragmas

- *Pragmas* are a preprocessor mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes (not covered in 61C)
- Good mechanism for OpenMP because compilers that don't recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas

`parallel` Pragma and Scope

- Basic OpenMP construct for parallelization:

```
#pragma omp parallel
{
    /* code goes here */
}
```

This is annoying, but curly brace MUST go on separate line from #pragma

  – *Each* thread runs a copy of code within the block
  – Thread scheduling is *non-deterministic*
- OpenMP default is *shared* variables
  – To make private, need to declare with pragma:

```
#pragma omp parallel private (x)
```

Thread Creation

- How many threads will OpenMP create?
- Defined by OMP_NUM_THREADS environment variable (or code procedure call)
  - Set this variable to the *maximum* number of threads you want OpenMP to use
- Usually equals the number of cores in the underlying hardware on which the program is run

OMP_NUM_THREADS

- Shell command to set number threads:

```
export OMP_NUM_THREADS=x
```

- Shell command check number threads:

```
echo $OMP_NUM_THREADS
```

- OpenMP intrinsic to set number of threads:

```
omp_set_num_threads(x);
```

- OpenMP intrinsic to get number of threads:

```
num_th = omp_get_num_threads();
```

- OpenMP intrinsic to get Thread ID number:

```
th_ID = omp_get_thread_num();
```

Parallel Hello World

```
#include <stdio.h>
#include <omp.h>
int main () {
  int nthreads, tid;
  /* Fork team of threads with private var tid */
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num(); /* get thread id */
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master and terminate */
}
```

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Directives
  - Workshare
  - Synchronization
- OpenMP Common Pitfalls
- Hardware: Transistors
- Bonus: `sections` Example

Administrivia

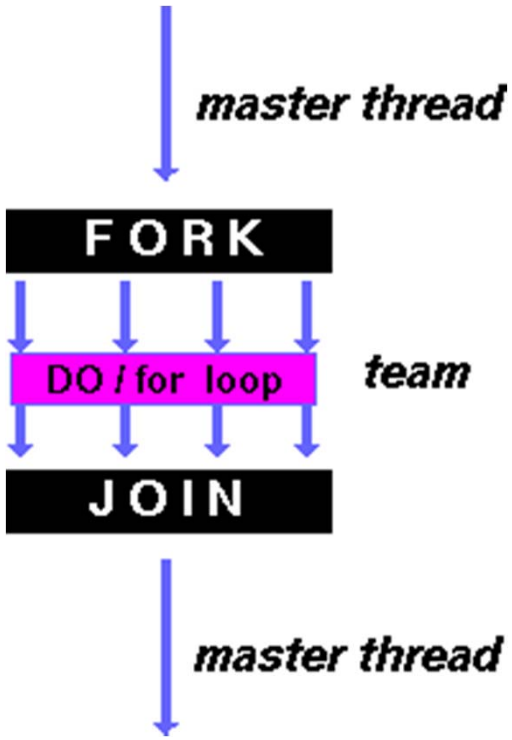
- Midterm re-grade requests due Thursday
- Project 2: Matrix Multiply Performance Improvement
  - Work in groups of two!
  - Part 1: Due July 22 (this Sunday)
  - Part 2: Due July 29
- HW 4 also due July 25

Agenda

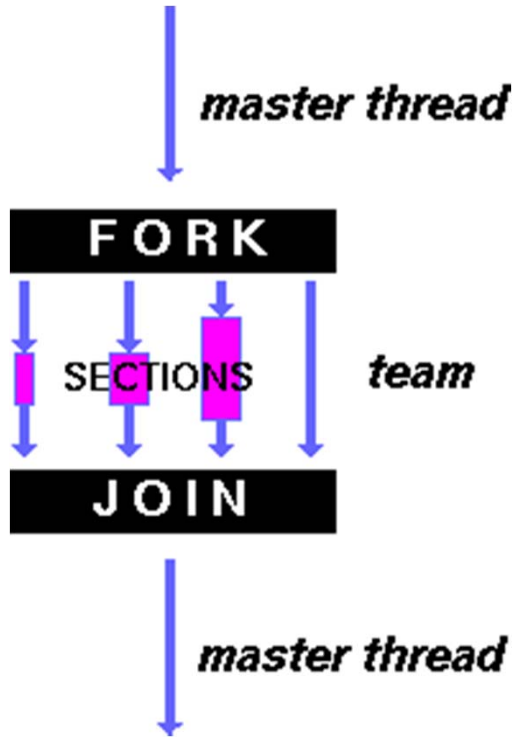
- OpenMP Introduction
- Administrivia
- OpenMP Directives
  - Workshare
  - Synchronization
- OpenMP Common Pitfalls
- Hardware: Transistors
- Bonus: `sections` Example

OpenMP Directives (Work-Sharing)

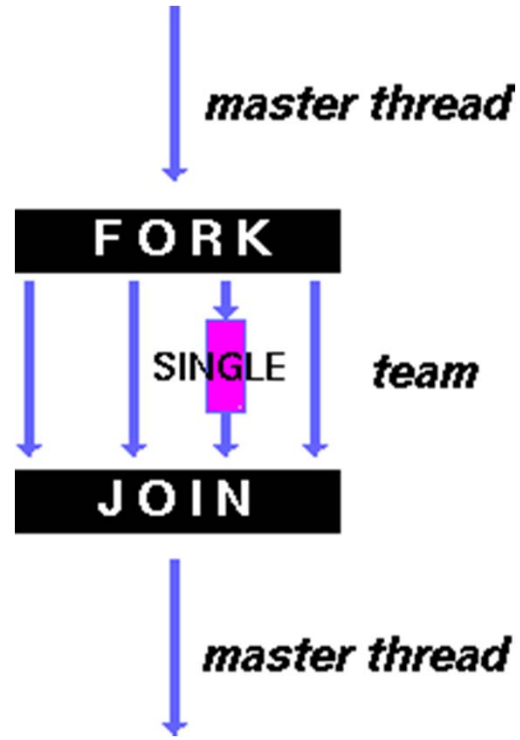
- These are defined *within* a `parallel` section



Shares iterations of a loop across the threads



Each section is executed by a separate thread



Serializes the execution of a thread

Parallel Statement Shorthand

```
#pragma omp parallel
{
  #pragma omp for
  for(i=0;i<len;i++) { … }
}
```

This is the only directive in the parallel section

can be shortened to:

```
#pragma omp parallel for
  for(i=0;i<len;i++) { … }
```

- Also works for `sections`

Building Block: `for` loop

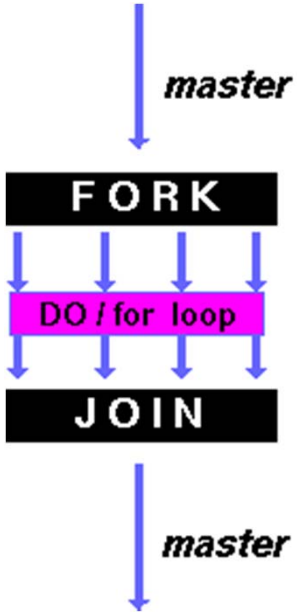
```
for (i=0; i<max; i++) zero[i] = 0;
```

- Break *for loop* into chunks, and allocate each to a separate thread
  - e.g. if `max` = 100 with 2 threads:
    assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple "shape" for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed ← In general, don't jump outside of any pragma block
  - i.e. No `break`, `return`, `exit`, `goto` statements

Parallel `for` *pragma*

```
#pragma omp parallel for
  for (i=0; i<max; i++) zero[i] = 0;
```

- Master thread creates additional threads, each with a separate execution context
- All variables declared outside for loop are shared by default, except for loop index which is *private* per thread (Why?)
- Implicit synchronization at end of for loop
- Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, …, (max/n)-1;
  - Thread 1 gets max/n, max/n+1, …, 2*(max/n)-1
  - Why?



OpenMP Timing

- Elapsed wall clock time:

```
double omp_get_wtime(void);
```

  - Returns elapsed wall clock time in seconds
  - Time is measured per thread, no guarantee can be made that two distinct threads measure the same time
  - Time is measured from "some time in the past," so subtract results of two calls to `omp_get_wtime` to get elapsed time

Matrix Multiply in OpenMP

```
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, i, j, k)
  for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
      tmp = 0.0;
      for( k=0; k<Pdim; k++){
        /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
        tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));
      }
      *(C+(i*Ndim+j)) = tmp;
    }
  }
run_time = omp_get_wtime() - start_time;
```

Outer loop spread across N threads; inner loops inside a single thread

Notes on Matrix Multiply Example

- More performance optimizations available:
  - Higher *compiler optimization* (-O2, -O3) to reduce number of instructions executed
  - *Cache blocking* to improve memory performance
  - Using SIMD SSE3 instructions to raise floating point computation rate (*DLP*)

OpenMP Directives (Synchronization)

- These are defined *within* a `parallel` section
- `master`
  - Code block executed only by the master thread (all other threads skip)
- `critical`
  - Code block executed by only one thread at a time
- `atomic`
  - Specific memory location must be updated atomically (like a mini-`critical` section for writing to memory)
  - Applies to single statement, not code block

OpenMP Reduction

- *Reduction* specifies that one or more private variables are the subject of a reduction operation at end of parallel region
  - Clause `reduction(operation:var)`
  - *Operation:* Operator to perform on the variables at the end of the parallel region
  - *Var:* One or more variables on which to perform scalar reduction

```
#pragma omp for reduction(+:nSum)
  for (i = START ; i <= END ; i++)
    nSum += i;
```

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Directives
  - Workshare
  - Synchronization
- OpenMP Common Pitfalls
- Hardware: Transistors
- Bonus: `sections` Example

OpenMP Pitfall #1: Data Dependencies

- Consider the following code:

```
a[0] = 1;
for(i=1; i<5000; i++)
  a[i] = i + a[i-1];
```

- There are dependencies between loop iterations!
  - Splitting this loop between threads does not guarantee in-order execution
  - Out of order loop execution will result in undefined behavior (i.e. likely wrong result)

Open MP Pitfall #2: Sharing Issues

- Consider the following loop:

```
#pragma omp parallel for
        for(i=0; i<n; i++){
                temp = 2.0*a[i];
                a[i] = temp;
                b[i] = c[i]/temp;
        }
```

- temp is a shared variable!

```
#pragma omp parallel for private(temp)
        for(i=0; i<n; i++){
                temp = 2.0*a[i];
                a[i] = temp;
                b[i] = c[i]/temp;
        }
```

OpenMP Pitfall #3: Updating Shared Variables Simultaneously

- Now consider a global sum:

```
for(i=0; i<n; i++)
      sum = sum + a[i];
```

- This can be done by surrounding the summation by a `critical/atomic` section or `reduction` clause:

```
#pragma omp parallel for reduction(+:sum)
{
   for(i=0; i<n; i++)
         sum = sum + a[i];
}
```

  – Compiler can generate highly efficient code for `reduction`

OpenMP Pitfall #4: Parallel Overhead

- Spawning and releasing threads results in significant overhead
- Better to have fewer but larger parallel regions
  - Parallelize over the largest loop that you can (even though it will involve more work to declare all of the private variables and eliminate dependencies)

OpenMP Pitfall #4: Parallel Overhead

```
start_time = omp_get_wtime();
for (i=0; i<Ndim; i++){
  for (j=0; j<Mdim; j++){
    tmp = 0.0;
    #pragma omp parallel for reduction(+:tmp)
      for( k=0; k<Pdim; k++){
        /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
        tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));
      }
    *(C+(i*Ndim+j)) = tmp;
  }
}
run_time = omp_get_wtime() - start_time;
```

Too much overhead in thread generation to have this statement run this frequently.

Poor choice of loop to parallelize.

Get To Know Your Staff

- Category: **Television**

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Directives
  - Workshare
  - Synchronization
- OpenMP Common Pitfalls
- Hardware: Transistors
- Bonus: `sections` Example

Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C)

Compiler

Assembly Language Program (e.g. MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)

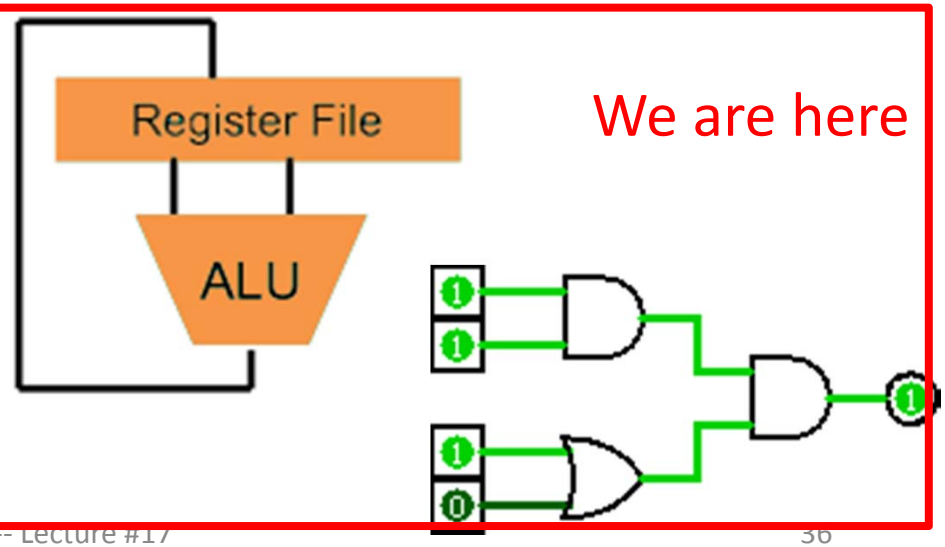
Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

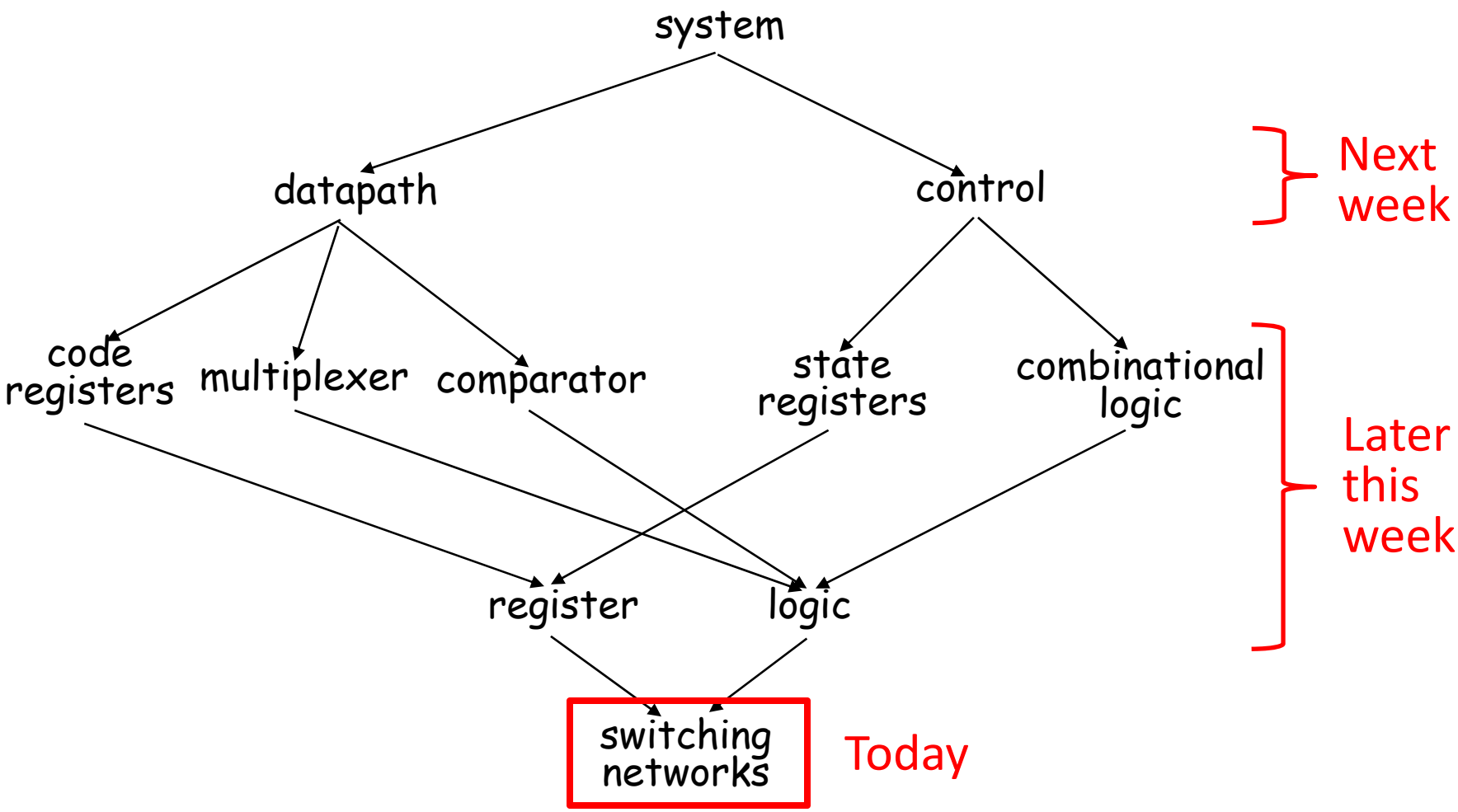


We are here

Hardware Design

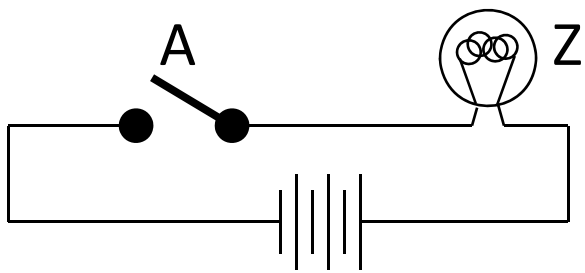
- **Upcoming:** We'll study how a modern processor is built, starting with basic elements as building blocks
- Why study hardware design?
  - Understand capabilities and limitations of hardware in general and processors in particular
  - What processors can do fast and what they can't do fast (avoid slow things if you want your code to run fast!)
  - Background for more in-depth courses (CS150, CS152)
  - You may need to design own custom hardware for extra performance (some commercial processors today have customizable hardware)

Design Hierarchy

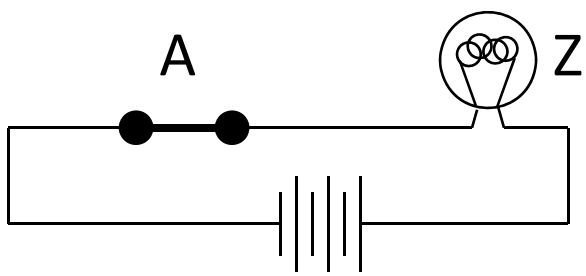


Switches (1/2)

- The basic element of physical implementations
- Convention: if input is a "1," the switch is *asserted*



Open switch if A is "0" (unasserted) and turn OFF light bulb (Z)



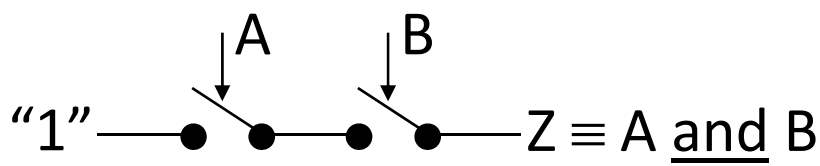
Close switch if A is "1" (asserted) and turn ON light bulb (Z)

In this example, Z ≡ A.

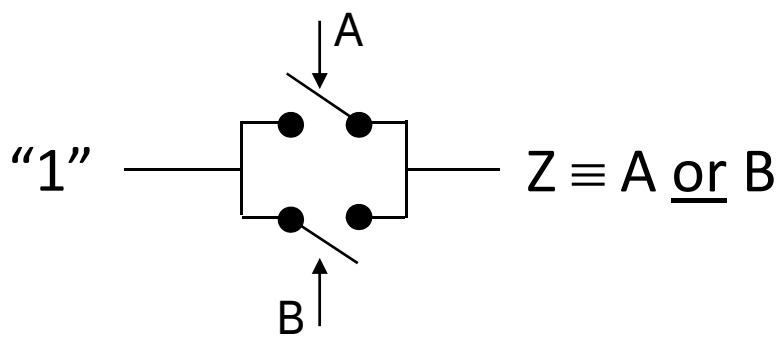
Switches (2/2)

- Can compose switches into more complex ones (Boolean functions)
  - Arrows show action upon assertion (1 = close)

AND:

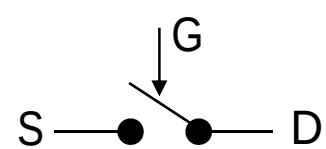


OR:



Transistor Networks

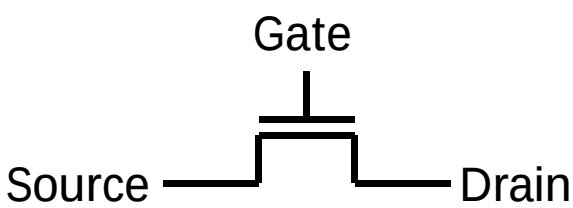
- Modern digital systems designed in CMOS
  - MOS: Metal-Oxide on Semiconductor
  - C for *complementary*: use pairs of normally-open and normally-closed switches
- CMOS transistors act as voltage-controlled switches
  - Similar, though easier to work with, than relay switches from earlier era
  - Three terminals: **S**ource, **G**ate, and **D**rain



CMOS Transistors

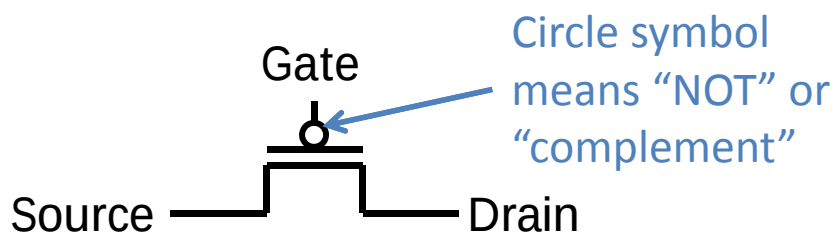
- Switch action based on *relative* voltages on Gate (V_G) and Source (V_S) terminals
  - Threshold voltage (V_T) determines whether of not Source and Drain terminals are connected
  - When not connected, Drain left "floating"

N-channel Transistor



$V_G – V_S < V_T$: Switch OPEN
$V_G – V_S > V_T$: Switch CLOSED

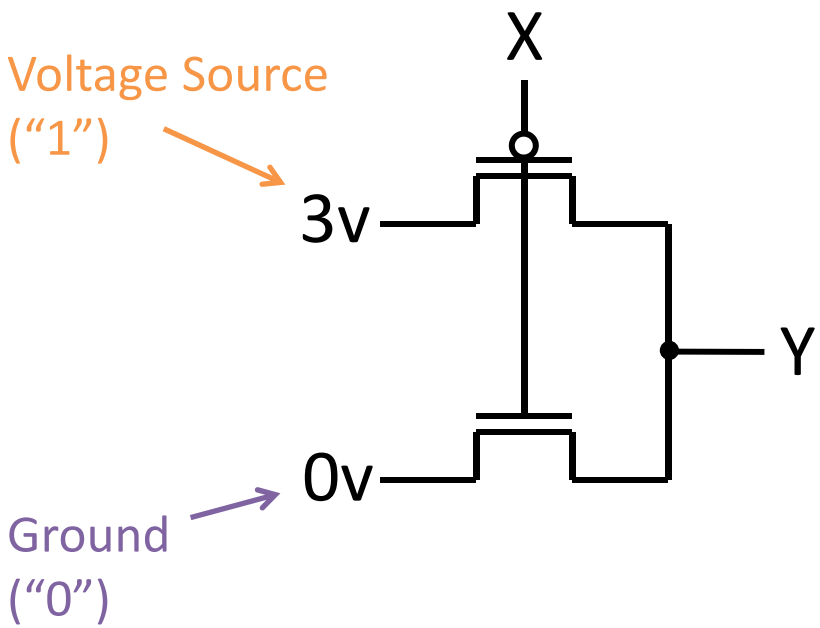
P-channel Transistor



$V_G – V_S < V_T$: Switch CLOSED
$V_G – V_S > V_T$: Switch OPEN

MOS Networks

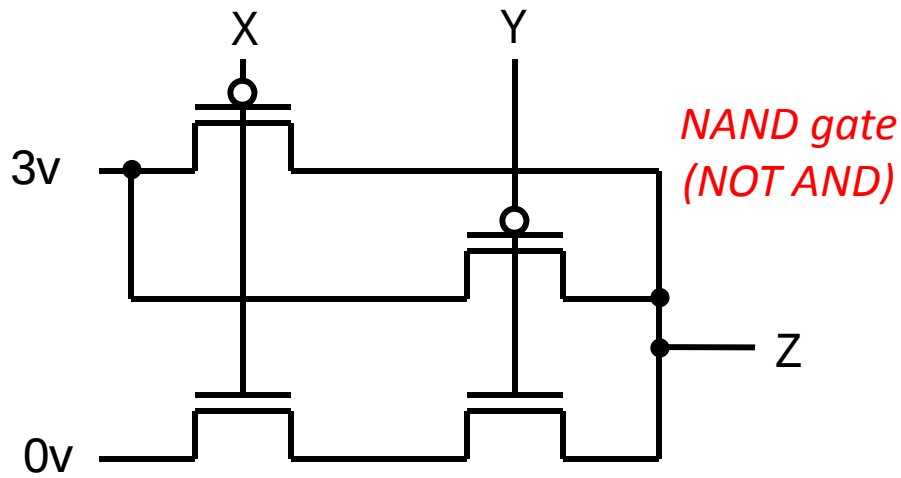
- What is the relationship between X and Y?



X	Y
0 volts	3 volts
3 volts	0 volts

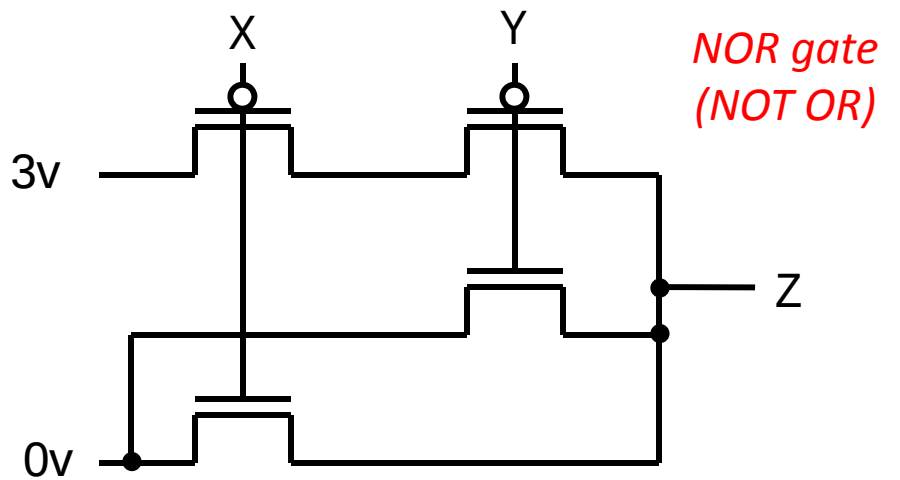
Called an inverter or NOT gate

Two Input Networks



NAND gate (NOT AND)

X	Y	Z
0 volts	0 volts	3 volts
0 volts	3 volts	3 volts
3 volts	0 volts	3 volts
3 volts	3 volts	0 volts



NOR gate (NOT OR)

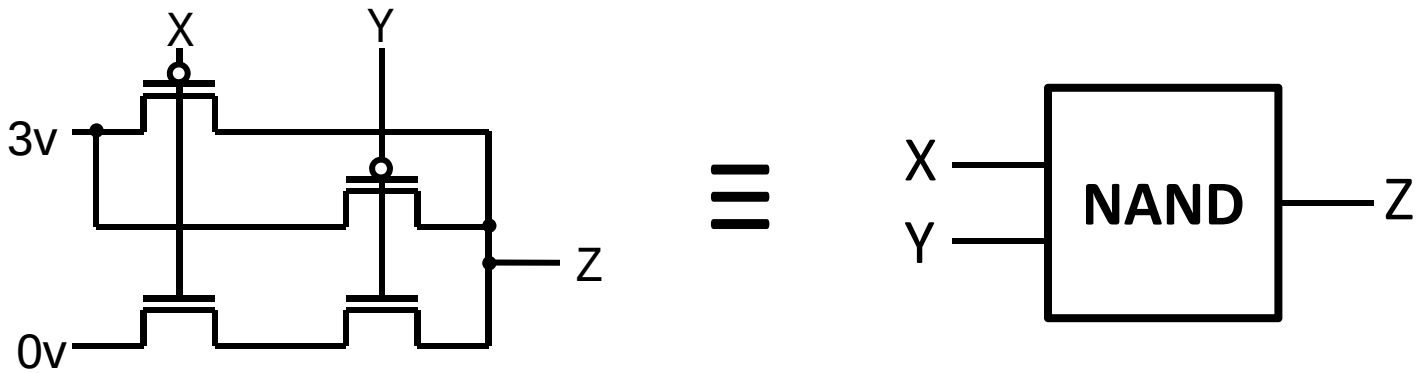
X	Y	Z
0 volts	0 volts	3 volts
0 volts	3 volts	0 volts
3 volts	0 volts	0 volts
3 volts	3 volts	0 volts

Transistors and CS61C

- The internals of transistors are important, but won't be covered in this class
  - Better understand Moore's Law
  - Physical limitations relating to speed and power consumption
  - Actual physical design & implementation process
  - Can take EE40, EE105, and EE140
- We will proceed with the abstraction of *Digital Logic* (0/1)

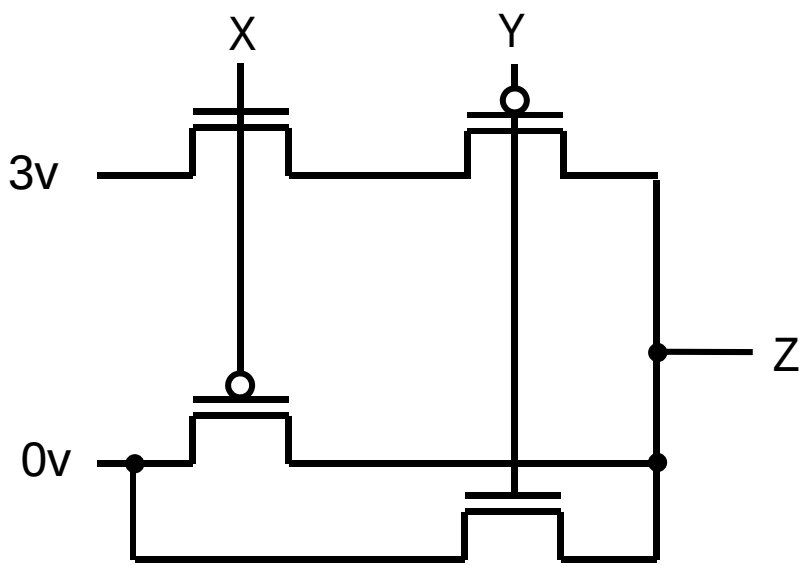
Block Diagrams

- In reality, chips composed of just transistors and wires
  - Small groups of transistors form useful building blocks, which we show as *blocks*



- Can combine to build higher-level blocks
  - You can build AND, OR, and NOT out of NAND!

Question: Which set(s) of inputs will result in the output Z being 3 volts?



	X	Y
☐	0	0
☐	0	1
☐	1	0
☐	1	1

Summary

- OpenMP as simple parallel extension to C
  - During parallel fork, be aware of which variables should be shared vs. private among threads
  - Work-sharing accomplished with for/sections
  - Synchronization accomplished with critical/atomic/reduction
- Hardware is made up of transistors and wires
  - Transistors are voltage-controlled switches
  - Building blocks of all higher-level blocks

BONUS SLIDES

You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture.

They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Directives
  - Workshare
  - Synchronization
- OpenMP Common Pitfalls
- Hardware: Transistors
- Bonus: `sections` Example

Workshare `sections` Example (1/4)

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

#define N     50

int main (int argc, char *argv[]) {
  int i, nthreads, tid;
  float a[N], b[N], c[N], d[N];

  /* Initialize arrays with some data */
  for (i=0; i<N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
    c[i] = d[i] = 0.0;
  }

/* main() continued on next slides... */
```

Workshare `sections` Example (2/4)

```
#pragma omp parallel private(i,tid) {
  tid = omp_get_thread_num();
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
  }
  printf("Thread %d starting...\n",tid);
  #pragma omp sections nowait {
    #pragma omp section {
      printf("Thread %d doing section 1\n",tid);
      for (i=0; i<N; i++){
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
      }
    }
```

Declare Sections

Define individual sections

Workshare `sections` Example (3/4)

```
      #pragma omp section {
        printf("Thread %d doing section 2\n",tid);
        for (i=0; i<N; i++){
          d[i] = a[i] * b[i];
          printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
        }
      }
    }  /* end of sections */
    printf("Thread %d done.\n",tid);
  }    /* end of parallel section */
}      /* end of main */
```

2nd section

Workshare `sections` Example (4/4)

```
Number of threads = 2
Thread 1 starting...
Thread 0 starting...
Thread 1 doing section 1
Thread 0 doing section 2
Thread 1: c[0]= 22.350000
Thread 0: d[0]= 0.000000
Thread 1: c[1]= 24.850000
Thread 0: d[1]= 35.025002
Thread 1: c[2]= 27.350000
Thread 0: d[2]= 73.050003
Thread 1: c[3]= 29.850000
Thread 0: d[3]= 114.075005
Thread 1: c[4]= 32.349998
Thread 0: d[4]= 158.100006
Thread 1: c[5]= 34.849998
Thread 0: d[5]= 205.125000
Thread 1: c[6]= 37.349998
Thread 0: d[6]= 255.150009
Thread 1: c[7]= 39.849998
Thread 0: d[7]= 308.175018
Thread 0: d[8]= 364.200012
Thread 0: d[9]= 423.225006
Thread 0: d[10]= 485.249969
Thread 0: d[11]= 550.274963
Thread 0: d[12]= 618.299988
Thread 0: d[13]= 689.324951
Thread 0: d[14]= 763.349976
Thread 0: d[15]= 840.374939
```

Notice inconsistent thread scheduling!

```
. . .
Thread 1: c[33]= 104.849998
Thread 0: d[41]= 3896.024902
Thread 1: c[34]= 107.349998
Thread 0: d[42]= 4054.049805
Thread 1: c[35]= 109.849998
Thread 0: d[43]= 4215.074707
Thread 1: c[36]= 112.349998
Thread 0: d[44]= 4379.100098
Thread 1: c[37]= 114.849998
Thread 1: c[38]= 117.349998
Thread 1: c[39]= 119.849998
Thread 1: c[40]= 122.349998
Thread 1: c[41]= 124.849998
Thread 1: c[42]= 127.349998
Thread 1: c[43]= 129.850006
Thread 1: c[44]= 132.350006
Thread 0: d[45]= 4546.125000
Thread 1: c[45]= 134.850006
Thread 0: d[46]= 4716.149902
Thread 1: c[46]= 137.350006
Thread 0: d[47]= 4889.174805
Thread 1: c[47]= 139.850006
Thread 0: d[48]= 5065.199707
Thread 0: d[49]= 5244.225098
Thread 0 done.
Thread 1: c[48]= 142.350006
Thread 1: c[49]= 144.850006
Thread 1 done.
```

`nowait` option in sections allows threads to exit "early"