More On Synchronization And Threading
Reminder:
ILP vs. TLP

- **Instruction Level Parallelism**
  - Multiple instructions in execution at the same time, e.g., instruction pipelining
  - Superscalar: launch more than one instruction at a time, typically from one instruction stream
  - ILP limited because of pipeline hazards
ILP vs. TLP

- **Thread Level Parallelism**
  - *Thread*: sequence of instructions, with own program counter and processor state (e.g., register file) but common memory within a process
  - *Process*: consists of at least one but could be arbitrary number of threads
    - Each process has its own memory space
  - **Multicore**:
    - Physical CPU: One thread (at a time) per CPU, in software OS switches threads typically in response to I/O events like disk read/write
    - Logical CPU: Fine-grain thread switching, in hardware, when thread blocks due to cache miss/memory access
    - Hyperthreading (aka Simultaneous Multithreading--SMT): Exploit superscalar architecture to launch instructions from different threads at the same time!
• SMT (Symmetric Multithreading/Intel Hyperthreading): Logical CPUs > Physical CPUs
  • Run multiple threads at the same time per core
  • Each thread has own architectural state (PC, Registers, etc.)
  • Share resources (cache, instruction unit, execution units)
  • Improves Core CPI (clock ticks per instruction)
  • May degrade Thread CPI (Utilization/Bandwidth v. Latency)
  • See http://dada.cs.washington.edu/smt/
Summary: Multithreaded Categories

<table>
<thead>
<tr>
<th>Time (processor cycle)</th>
<th>Superscalar</th>
<th>Fine-Grained</th>
<th>Coarse-Grained</th>
<th>Multiprocessing</th>
<th>Simultaneous Multithreading</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Thread 1
- Thread 2
- Thread 3
- Thread 4
- Thread 5
- Idle slot
OpenMP Building Block: for loop rather than just the parallel block

- **for** (i=0; i<max; i++) zero[i] = 0;
- Breaks 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:
    Not a good idea to be changing the loop bounds in the loop itself
- No premature exits from the loop allowed
  - i.e. No break, return, exit, goto statements, just simple **for** and **while** loops
OpenMP 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 *implicitly* private per thread

- Implicit “barrier” 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
Example 2: Computing \( \pi \)

Numerical Integration

Mathematically, we know that:

\[
\int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi
\]

We can approximate the integral as a sum of rectangles:

\[
\sum_{i=0}^{N} F(x_i) \Delta x \approx \pi
\]

Where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
Working Parallel $\pi$ without a for loop

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

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);

#pragma omp parallel
{
    int id = omp_get_thread_num();
    for (int i=id; i<num_steps; i+=NUM_THREADS) {
        double x = (i+0.5) *step;
        sum[id] += 4.0*step/(1.0+x*x);
        printf("i =%3d, id =%3d\n", i, id);
    }
}

double pi = 0;
for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
printf ("pi = %6.12f\n", pi);
```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
    
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        printf("i =%3d, id =%3d\n", i, id);
    }
    
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}

i =  1, id =  1
i =  0, id =  0
i =  2, id =  2
i =  3, id =  3
i =  5, id =  1
i =  4, id =  0
i =  6, id =  2
i =  7, id =  3
i =  9, id =  1
i =  8, id =  0
pi = 3.142425985001
Scale up: num_steps = $10^6$

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

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 1000000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
   omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            // printf("i =%3d, id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf("pi = %6.12f\n", pi);
}
```

$\pi = 3.141592653590$
Can We Parallelize Computing \texttt{sum}?

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

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
    
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) * step;
            sum[i] += 4.0*step/(1.0+x**x);
        }
        pi += sum[id];
    }

    printf ("pi = %6.12f\n", pi);
}
```

Always looking for ways to beat Amdahl’s Law ...

Summation inside parallel section
- Insignificant speedup in this example, but ...
- $\pi = 3.138450662641$
- Wrong! And value changes between runs?!
- What’s going on?
What’s Going On?

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

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/(double)num_steps;
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);

#pragma omp parallel
{
    int id = omp_get_thread_num();
    for (int i=id; i<num_steps; i+=NUM_THREADS) {
        double x = (i+0.5) *step;
        sum[id] += 4.0*step/(1.0+x*x);
    }
    pi += sum[id];
}
printf ("pi = %6.12f\n", pi);
```
OpenMP Reduction

- `double avg, sum=0.0, A[MAX]; int i;`  
  
  ```c
  #pragma omp parallel for private ( sum )
  for (i = 0; i <= MAX ; i++) {sum += A[i];}
  avg = sum/MAX;  // bug, we only get the master thread's sum
  ```

- Problem is that we really want sum over all threads!

- **Reduction**: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region:
  
  ```c
  reduction(operation:var) where
  ```
  
  - Operation: operator to perform on the variables (var) at the end of the parallel region
  - Var: One or more variables on which to perform scalar reduction: private than combined

- `double avg, sum=0.0, A[MAX]; int i;`  
  
  ```c
  #pragma omp for reduction(+ : sum)
  for (i = 0; i <= MAX ; i++) {sum += A[i];}
  avg = sum/MAX;
  ```
Calculating $\pi$ Simple Version

```c
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
    int i;    double  x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #pragma omp parallel private ( i, x )
    {
        int id = omp_get_thread_num();
        for (i=id, sum[id]=0.0; i<num_steps; i=i+NUM_THREADS)
        {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=1; i<NUM_THREADS; i++)
        sum[0] += sum[i];  pi = sum[0];
    printf ("pi = %6.12f\n", pi);
}
```
Version 2: parallel for, reduction

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

static long num_steps = 100000;
double step;

void main ()
{
    int i;    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=1; i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = sum;
    printf ("pi = %6.8f
", pi);
}
```
Reduction Options...

- Arithmetic
  - + * −
- Comparison
  - min max
- Logical
  - & && | || ^
OpenMP Timing

- Elapsed wall clock time:
  ```c
  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

```c

start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, j, k)
  for (i=0; i<M; i++){
    for (j=0; j<N; j++){
      tmp = 0.0;
      for (k=0; k<P; k++){
        /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/
        tmp += A[i][k] * B[k][j];
      }
      C[i][j] = tmp; // doing this to tmp prevents
                      // potential cache issues from multiple
                      // writers & "false sharing"
    }
  }
run_time = omp_get_wtime() - start_time;
```

Outer loop spread across N threads; inner loops inside a single thread
Matrix Multiply in Open MP

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

• Two memory accesses form a data race if from different threads access same location, at least one is a write, and they occur one after another

• If there is a data race, result of program varies depending on chance (which thread first?)

• Avoid data races by synchronizing writing and reading to get deterministic behavior

• Synchronization done by user-level routines that rely on hardware synchronization instructions
Reminder: Locks

• Computers use locks to control access to shared resources
  • Serves purpose of microphone in example
  • Also referred to as “semaphore”
  • Although "semaphores" have slightly different semantics

• Usually implemented with a variable
  • In most object-oriented languages its an object that you can call
  • Under the hood its usually just an integer that has atomic updates:
    0 == unlocked, 1 == locked

• Two operations:
  • lock.acquire() // blocks this thread until the lock is unlocked
  • lock.release() // Unlocks the lock

• OpenMP also has a lock
  • #pragma omp critical
    { 
      ... Only one thread is running at a time here
    }
Hardware Synchronization (aka building atomicity)

- Hardware support required to prevent an interloper (another thread) from changing the value
  - Atomic read/write memory operation
    - No other access to the location allowed between the read and write
- How best to implement in software?
  - Single instr? Atomic swap of register ↔ memory
  - Pair of instr? One for read, one for write
Synchronization in RISC-V option one: Read/Write Pairs

- **Load reserved:** \texttt{lr rd, rs1}
  - Load the word pointed to by \texttt{rs} into \texttt{rd}, and add a reservation: this is \texttt{my memory}!!!

- **Store conditional:** \texttt{sc rd, rs1, rs2}
  - Store the value in \texttt{rs2} into the memory location pointed to by \texttt{rs1}, if the reservation is still valid and write the status in \texttt{rd}
    - Returns 0 (success) if location has not been updated by anybody else since the \texttt{lr}
    - Returns nonzero (failure) if location has been updated by anyone else, write doesn't happen
Synchronization in RISC-V Example

• Atomic swap (to test/set lock variable)
  Exchange contents of register and memory:
  \( s4 \leftrightarrow \text{Mem}(s1) \)

```
try:
  lr t1, s1        #load reserved
  sc t0, s1, s4    #store conditional
  bne t0, x0, try  #loop if sc fails
  add s4, x0, t1   #load value in s4
```

sc would fail if another thread executes a store here
Test-and-Set

- In a single atomic operation:
  - **Test** to see if a memory location is set (contains a 1)
  - **Set** it (to 1) if it isn’t (it contained a zero when tested)
    - Otherwise indicate that the Set failed, so the program can try again
  - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions
- Useful for implementing lock operations
Test-and-Set in RISC-V using lr/sc

- Example: RISC-V sequence for implementing a T&S at (s1)
  ```
  li t2, 1
  Try: lr t1, s1
      bne t1, x0, Try
      sc t0, s1, t2
      bne t0, x0, Try
  Locked:
      # critical section
  Unlock:
      sw x0, 0(s1)
  ```

Idea is that not for programmers to use this directly, but as a tool for enabling implementation of parallel libraries
**Clickers:** Consider the following code when executed *concurrently* by two threads.

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

```
# *(s0) = 2
lw  t0, 0(s0)
addi t0, t0, 1
add  t0, t0, t0
sw  t0, 0(s0)
```

A: 6  
B: 14  
C: 6 or 14  
D: 6 or 14 or 30
Why else use lr/sc?

- It actually allows some lock-free consistent structures
  - So although RISC-V has two separate atomic operation options, both have their uses
- EG...
  - fail:
    ```
    lr  t0, s0,
    addi t0, t0, 1
    add  t0, t0, t0
    sc  t0, s0, t0
    bne t0, s0, fail
    ```
- Allows the clicker question to be consistent without having a full lock, instead just 1 additional instruction to check for success on store conditional
  - Additionally if the code in between is short and restricted in instructions, this is guaranteed to work always
RISC-V Alternative: Atomic Memory Operations

- Three instruction rtype instructions
  - Swap, and, add, or, xor, max, min
- Take the value **pointed to** by rs1
  - Load it into rd
  - Apply the operation to that value with rs2
  - store the result back to where rs1 is pointed to
- Note, annoyingly obscure memory semantics
More On These Alternatives...

- Atomic swap is easy
  - It allows locks (s0 is a pointer to the lock, if that contains 0 its unlocked, nonzero if locked):
    
    ```
    li t0 1
    loop: amoswap t0 s0 t0
    bne t0 x0 loop
    ```

- Atomic associative operations are slightly different
  - Goal is to enable merging elements:
    Represents a common programming paradigm of the reduction in e.g. OpenMP

- And this is why RISC-V has **two** sets of atomic operations:
  - Atomic register+memory enables common reductions in single instructions, and is guaranteed to *always* work
  - `lr/sc` enables multiple-instructions for some level of lock-free manipulation
Deadlock

- Deadlock: a system state in which no progress is possible because everything is locked waiting for something else
- Dining Philosopher’s Lawyers Problem:
  - Pontificate until the left fork is available; when it is, pick it up
  - Pontificate until the right fork is available; when it is, pick it up
  - When both forks are held, eat for a fixed amount of time
  - Then, put the right fork down
  - Then, put the left fork down
  - Repeat from the beginning
- Solution?
Limiting Parallelism

- Locks act to inhibit parallelism
  - Innately sequential regions -> Amdahl's law problem...
  - Python is "threaded" but not really:
    A global interpreter lock means other threads can be waiting on I/O, but (mostly) only one compute thread at a time
- Can try to limit the locks
  - Rather than locking everything have a different lock for each region...
    - But be careful, then its much easier to get into deadlock situations
- Or try to eliminate locks altogether
  - Thus, e.g. why Go's parallelism is not focused around locks, and RISC-V lr/sc instructions
(Chip) Multicore Multiprocessor

- SMP: (Shared Memory) Symmetric Multiprocessor
  - Two or more identical CPUs/Cores
  - Single shared coherent memory
Multiprocessor Key Questions

• Q1 – How do they share data?

• Q2 – How do they coordinate?

• Q3 – How many processors can be supported?
Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors/cores
- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
- Effectively all multicore computers today are SMP
- Q3 - Depends on the workload!
  - Most systems go with "Best available single core within constraints, duplicate that"
Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory

![Diagram of multiprocessor caches](image-url)
Shared Memory and Caches

• What if?
• Processors 1 and 2 read Memory[1000] (value 20)
Shared Memory and Caches

- Now:
  - Processor 0 writes Memory[1000] with 40

Problem?
Keeping Multiple Caches Coherent

- Architect’s job: shared memory
  => keep cache values **coherent**

- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies

- Write transactions from one processor, other caches “snoop” the common interconnect checking for tags they hold
  - Invalidate any copies of same address modified in other cache
How Does HW Keep $ Coherent?

• We already saw how to do this with just valid and dirty, but we can also think of it this way...

• Each cache tracks state of each block in cache:

  1. Shared: up-to-date data, other caches may have a copy (Valid Bit only is set)

  2. Modified: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date (Dirty bit is also set)
Two Optional Performance Optimizations of Cache Coherency via New States

- Each cache tracks state of each *block* in cache:

  3. **Exclusive**: up-to-date data, no other cache has a copy, OK to write, memory up-to-date
     - If any other cache reads this line the state becomes *shared*
     - Can provide the line if I'm faster than main memory
     - If this writes to this line, state becomes *modified* but I don't need to broadcast this when I do the write

  4. **Owner**: up-to-date data, other caches may have a copy (they must be in Shared state)
     - But this cache now supplies data on read instead of going to memory:
       Saves the need for a write back when somebody else reads the page
Name of This Common Cache Coherency Protocol: MOESI

- Memory access to cache is either
  - Modified (in cache)
  - Owned (in cache)
  - Exclusive (in cache)
  - Shared (in cache)
  - Invalid (not in cache)

Snooping/Snoopy Protocols
- e.g., the Berkeley Ownership Protocol
- Berkeley Protocol is a wikipedia stub!
Shared Memory and Caches

- Example, now with cache coherence
- Processors 1 and 2 read Memory[1000]
- Processor 0 writes Memory[1000] with 40
But An Alternate // Programming Paradigm...

Communicating Sequential Processes

- OpenMP has **very restrictive** parallelism
  - Really only good for parallelizing loops with an optional reduction step
    - And Amdahl's law therefore quickly rears its ugly head
- Raw threads is **very easy** to get wrong
  - Deadlock situations
- Enter CSP:
  - A way for different threads to efficiently communicate
- A good CSP language: Go (golang)
  - (Only going to cover this if there is time)
What is Go

- Language created at Google starting in 2007
- Primarily by a bunch of old Unix hands: Robert Griesemer, Rob Pike, and Ken Thompson
- 1.0 released in March 2012

- Language continues to evolve, but a commitment to backwards compatibility (so far)
  - A correct program written today will still work tomorrow
    - I’m looking at you, python 3....

- Mostly C-ish but...
  - Strong typing, no pointer arithmetic, lambdas, interfaces, garbage collection and...
  - Strong emphasis on concurrent computation
Good Go Resources

- The Go website:
  - https://golang.org/
- Especially useful: Effective Go:
  - A cheatsheet of programming idioms. Several example from this lecture stolen from there
  - https://golang.org/doc/effective_go.html
- When searching Google, ask for **golang**, not go
  - The language may be Go, but golang refers to the language too
- If I'm starting new code and I need to care about performance, scalability, or maintainability, I use Go
So Think of Go as:

- C's general structure & concepts
  - But with implied ;s and a garbage collector
- A better typing system with interfaces, slices, and maps
  - No class inheritance, however
- Much more symmetric functions
  - Can return multiple values
- Scheme-like lexical scope
  - Lambdas and interior function declarations
- Communicating Synchronous Processes (CSP) concurrency
  - Multiple things at once in the same shared memory space: quite suitable for MIMD
Coroutines, err, goroutine

- Conceptually, a goroutine is just a thread...
  - `go fubar() // Executes fubar as a new coroutine`
- But in practice it is designed to be much lighter weight
  - Threads (e.g. in OpenMP) relatively expensive to create: Operating System involvement is never cheap
- Go's runtime instead pre-creates a series of threads
  - And then schedules the active coroutines itself to the available threads
- Result is goroutines are cheap
  - It is only slightly more expensive than a plain function call: new goroutine just has a small independent stack context switching between goroutines is very cheap
Channels

• Channels are the primary synchronization mechanism

• You have locks but why bother?

• A typed and (optionally) buffered communication channel

• $c := \text{make(chan int)}$
  $e := \text{make(chan fubar, 100)}$

• Writing data to a channel:

• $c \leftarrow 32$ // Writing blocks if unbuffered or full

• Reading from a channel:

• $\text{var } f = \leftarrow e$ // Reading blocks if no data
Channels as Synchronization Barriers

- Any writes in the code before writing to the channel complete first
  - `globalA = ...`
    - `d <- 1`  // The write to `globalA` must complete just before this
- Any reads in the code after reading from the channel do not start until channel-read takes place
  - `<- d`
    - `fubar(globalA)`  // Won't read `globalA` before channel read
- Otherwise, compiler can reorder **however it wants** as long as sequential semantics are preserved for the sequential function
  - Go may have removed a lot of ways to shoot yourself in the foot...
    - but unless you use channels you will easily blow it off with race conditions
Without Synchronization Barriers, The Compiler Can Go To Town

• This doesn't work!
  • Compiler can reorder the writes between x and done safely
    • Similar variants also possible

• This is one of the two biggest pitfalls of Go:
  • Unless you explicitly synchronize, multiple processes can write in "weird" ways
  • The other is the abysmal error/exception handling mechanism

```go
var x : string
var done : bool
func foo(){
    ...
    x = "something"
    done = true
}

func bar(){
    go foo()
    for !done {...}
    y := x
}
```
Using goroutines

• For things which may block or wait
  • EG, on input/output, waiting for stuff to happen, etc
  • Just create as many as you want!
    • Its cheap so why not: let the scheduler do useful work when another one is waiting
    • EG, if building a webserver, you launch a goroutine to handle each communication stream

• For performance tasks
  • Only create as many as there are CPU cores
    • Otherwise you are wasting resources
  • But it should be more efficient than OpenMP:
    • Thread creation is significantly more overhead than go
Select

- Select allows you to wait on multiple channels
  - `select {
      case c <- x:
          x, y = y, x+y
      case d := <- e:
          fmt.Println("Got %v", d)
      default:
          time.Sleep(50 * time.Millisecond)
  }`

- If can write or read to a channel, do so and **then** execute the associated case
  - If multiple cases are valid, choose one **at random**

- If nothing is available, execute default (if any)
  - If no default, just block until you can write or read
  - Default enables non-blocking read & write
Lots of other features for code correctness

- Compiler is, umm, persnickety
  - It is an error to declare but not use a variable or include but not use a package
- Designed to turn comments into documents
  - Including examples
- Libraries for building example and test routines
- Built in package management
- “Single workspace” notion
  - Use common modules to prevent code drift