Los Alamos

NATIONAL LABORATORY

EST. 1943

NNSA

National Nuclear Security Administration

TOP 500
The List.

Los Alamos
NATIONAL LABORATORY
EST. 1943

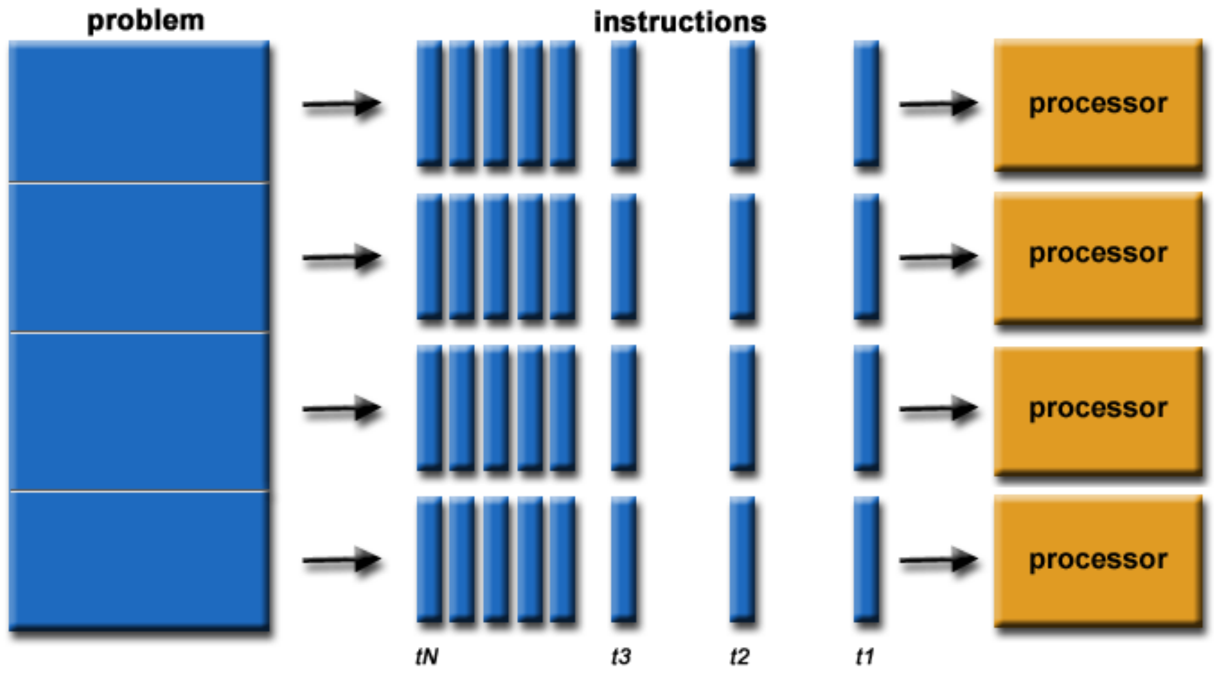
Delivering science and technology
to protect our nation
and promote world stability

# Parallel Programming

Presented by CSCNSI

## Working in parallel

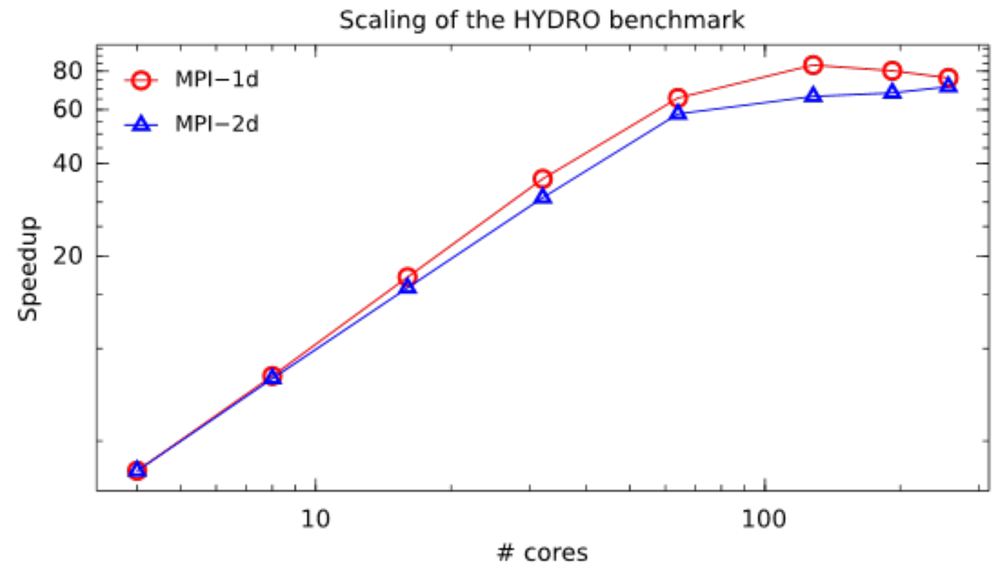
- Processors aren’t getting faster
- But we are getting more of them
- We have to find ways to work in parallel
  - Performance tuning = parallelization
- Modern optimization is largely the process of making algorithms more parallel

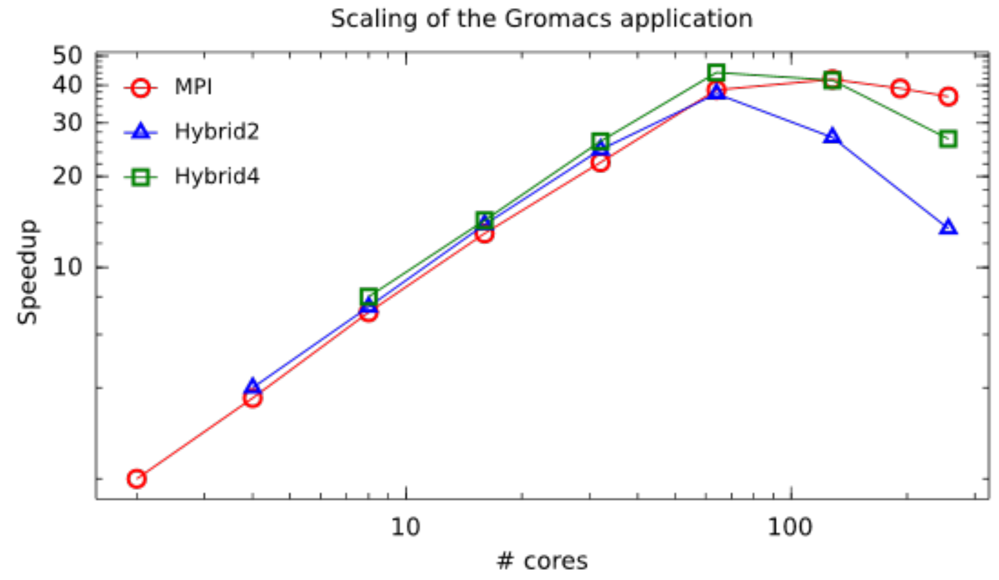


https://computing.llnl.gov/tutorials/parallel_comp/

## Parallel Scaling

- Scaling is hard
- Usually we reach a plateau
- Things that can get us over the plateau (roughly in order of efficacy)
  - Rethinking our algorithms
    - Sometimes requires rethinking scientific approximations
  - Writing better parallelism
  - Improving hardware level paralellism

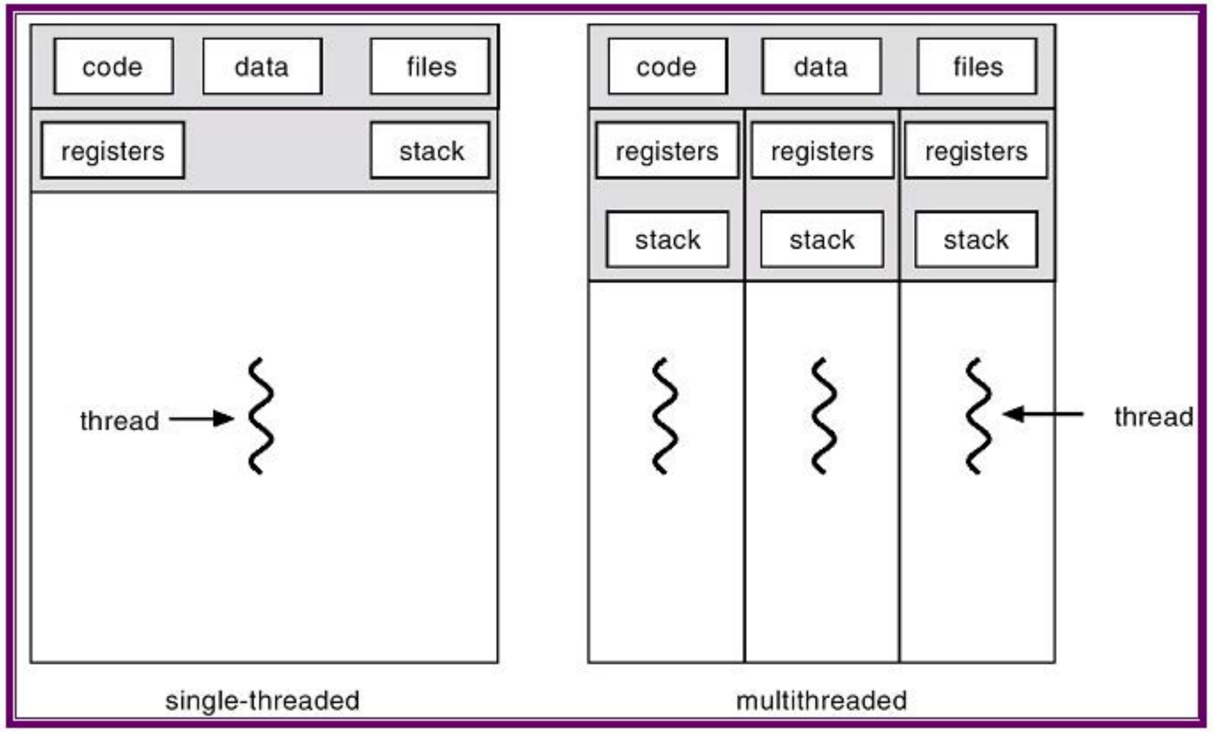




http://www.prace-ri.eu/best-practice-guide-knights-landing-january-2017/#scaling-and-speedup-NPB

## Threads

- Threads are lightweight, possibly parallel processes
- They ***share*** their memory (except stack)
- If done well, threads lead to huge improvements on multi-processor hardware
- If done poorly, threads can cost more than they save
  - ...and cause huge headaches.



http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html

## Structure of a typical threaded program

1. Create a thread that is set to run a defined function at start ("create")
2. Start the treads ("start")
   1. The threads to work
   2. The main thread can do work too
3. Wait for threads to finish ("join")

- The threads often call the same function
  - but do different work based on ID

Psuedo-code for threading structure

```
function runner() {
    id = get_thread_id()
    do_some_work(id)
}

function main() {
    t1 = pthread_create(runner)
    t2 = pthread_create(runner)
    t1.start()
    t2.start()
    /* do something */
    t1.join()
    t2.join()
}
```

## Race conditions

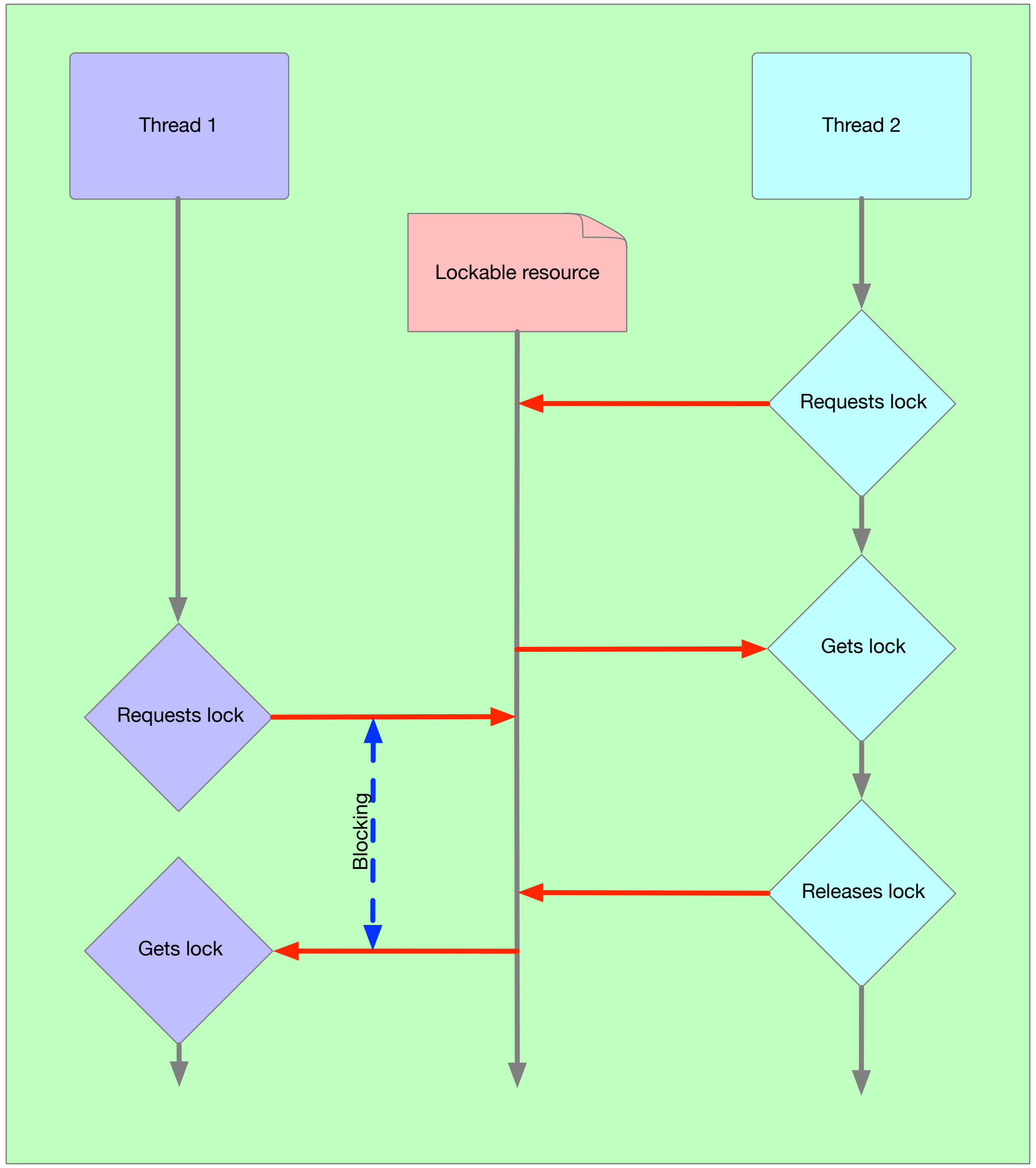
- Suppose t1's job involved writing data to a the file *data*
- While t1 is doing this, t2 is computing some data to add to the file
- After computing t2 reads the file and uses that data to finish its work
- *Usually* t2 takes long enough that *data* is ready
  - *...but what if it's not!*

- **A race condition exists when success of an algorithm depends on one thread/process "winning the race" against another.**

## Locking: Overview

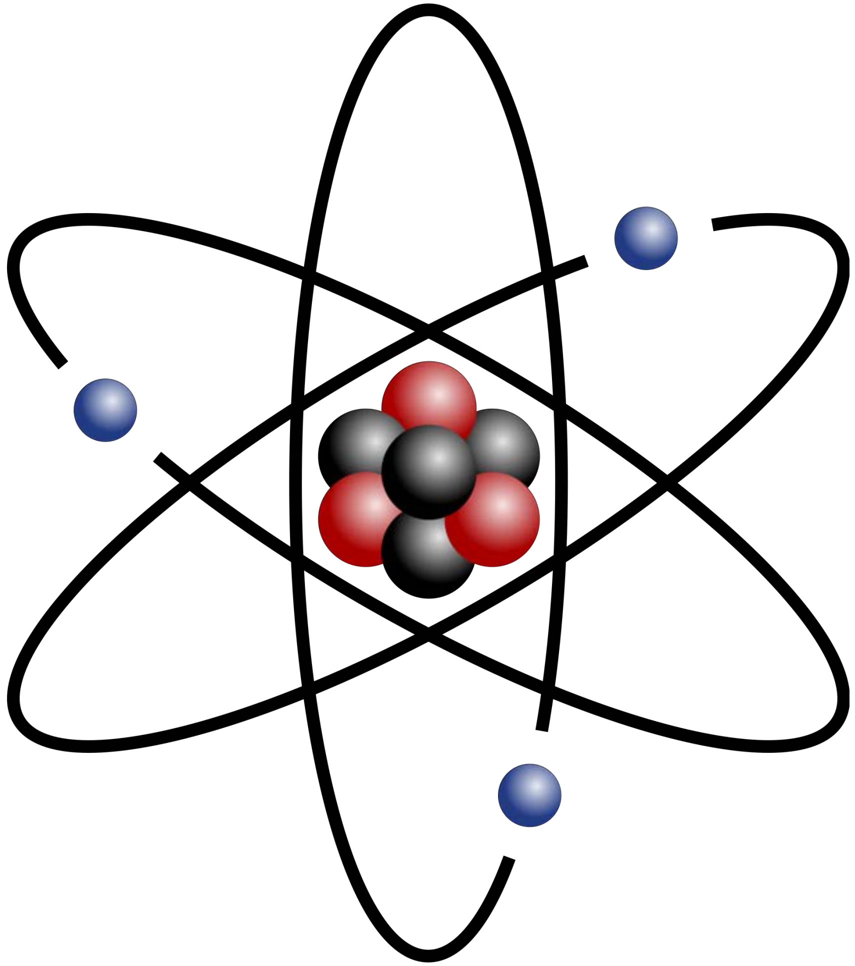
- How do we deal with things like race conditions?
- In our example, if:
  - t1 had a way to maintain exclusive access to *data*
  - ...and t2 would wait until that access was dropped.
    - Then there wouldn't be a race.

- This is an example of Locking.
- Locking can be used to synchronize concurrent computing processes/threads.



## Atomic operations

- An “atomic” operation is guaranteed to complete without interference from any concurrent process.
- Typically, this means they take exactly one instruction cycle on the CPU, making them uninterruptable.
- Atomic operations are the necessary basis for locking & synchronization
  - If we want to lock something, we need to know the the operation of locking isn’t itself a race
    - ...what if two things call lock at the same time?


https://en.wikipedia.org/wiki/Atomic_mass

## Locking: Mutex

- A Mutex has two operations:
  - Lock()
  - Unlock()
- Stores a reference in shared memory
- Uses atomic operations to read/set whether the lock is held
- If you try to Lock() an already locked mutex
  - ...you'll block until it gets Unlock()ed by someone else

https://medium.com/@the1mills/a-better-mutex-for-node-js-4b4897fd9f11

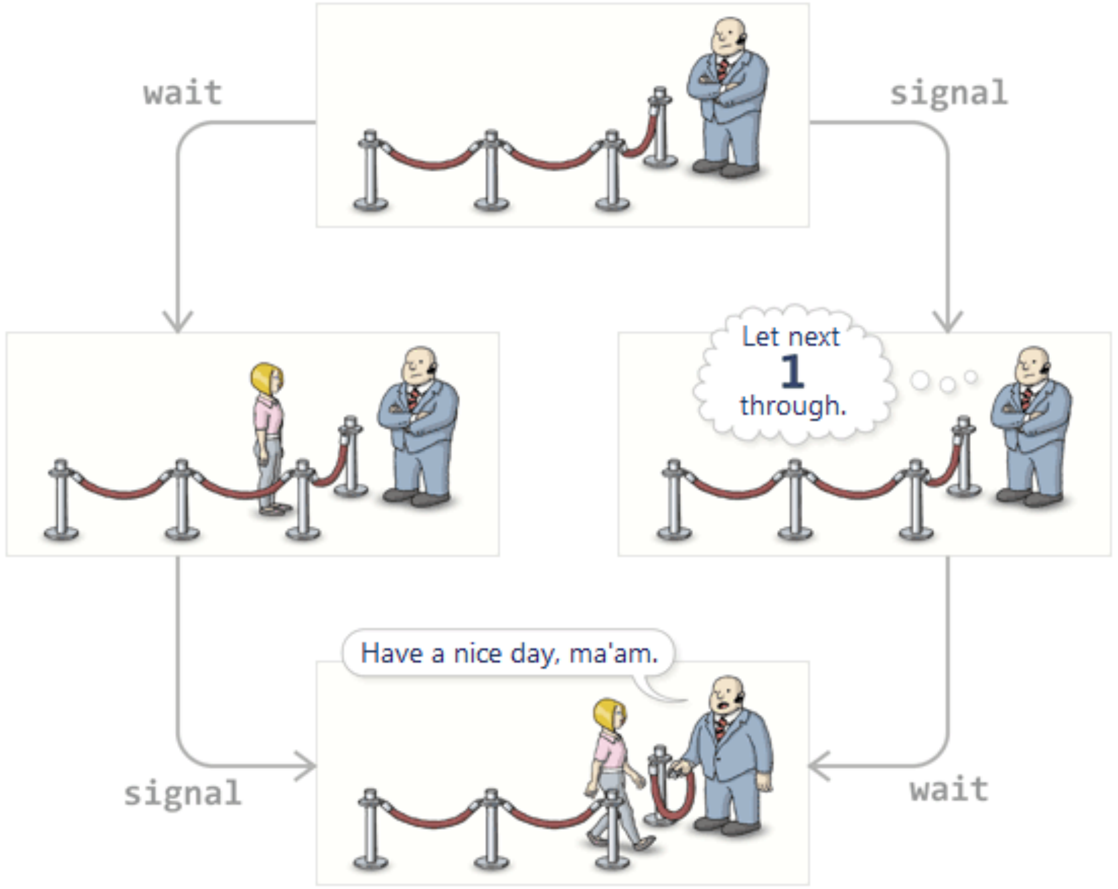
## Locking: RWMutex

- An RWMutex works like Mutex
- But:
  - There can be any number of RLocks() (read)
  - There can be only one WLock() (write)
  - WLock() and RLock() are mutually exclusive
  - Once WLock() is requested, new RLocks() block
    - ...once all RLock()s RUnlock(), WLock() is granted.

## Locking: Semaphore

- Two operations, and an integer:
  - Wait() decrements the integer by 1
    - And waits to be signaled
  - Signal() incriments the integer by 1
    - If the integer goes from < 0 to >= 0, waiting processes are signaled

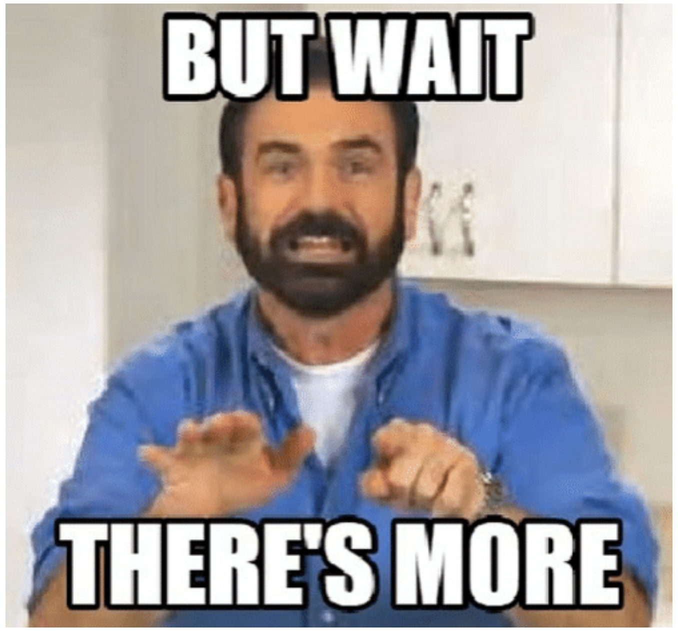
- Like Mutex, based on simple atomic add/subtract atomic operations.
- Can be used to implement things like RWLocks



https://preshing.com/20150316/semaphores-are-surprisingly-versatile/

## Locking: ...but wait! There’s more...

- Barriers: everybody must pause until we all call Barrier (synchronize).
- Conditional locks: lock until a condition is met.
- SpinLock: Like locking, but instead of blocking, pole for the lock.
- Event Locks: Block until this event is raised.



## Deadlocks

- Example:
  1. run1() locks t and calls run2()
  2. run2() tries to lock t, and blocks
     - run2 never exits
     - run1 never continues to t.Unlock()

- This is a *deadlock*.
- A deadlock is a locking condition that has circular dependencies.
- Mostly avoided by careful locking symantics.

Psuedo-code for deadlock

```
function run1() {
    t.Lock()
    run2()
    t.Unlock()
}

function run2() {
    t.Lock()
    do_stuff()
    t.Unlock()
}
```

## Threads aren't the only way

- Multiple processes with inter-process communication
  - Resource expensive, but this is mattering less and less
  - Often use UNIX domain sockets for communication (a special kind of file)
- Thread abstraction layers:
  - OpenMP—C/C++ extensions that give simple parallelism
  - Goroutines—Specific to Golang, message passing between lightweight concurrent routines
  - Event-driven design—register handlers for events, raise events.  Run handlers concurrently.  This is actually how the kernel works.
  - Higher-level abstractions, e.g. Python work queues
- No shared memory at all, but message passing:
  - E.g. MPI (next time)

## OpenMP: Overview

- Standard implemented as “pragmas” (compiler directives)
  - Can be turned on/off for flexible coding & debugging
  - Commonly included within the compiler itself
  - Can vary the number of threads without recoding

- Typically wraps an existing loop, and makes it run in parallel

Psuedo-code for OpenMP structure

```
#include <omp.h>
int main() {
     int cpu_id = -1;
     cpu_id = sched_getcpu( );
...
     #pragma omp parallel for
     for (int i = 0; i < 4; i++){
          printf(“value:%d,
                    thread:%d\n”, i,
                    omp_get_thread_num());
     }
...
}
```

## MPI

## What is MPI?

- “Message Passing Interface”
- A pluggable transport layer for sending bits of data to different processes
- Supports optimized transports for high speed fabrics (e.g. Infiniband)
- Allows efficient, fast inter-process communication across a cluster
- Allows easy porting between clustered platforms

## MPI: History

- 1993 - Draft for standard submitted at supercomputing
- 1994 – MPI 1.0
- 1998 – MPI-2
- 2012 – MPI-3
- MPI-4 – Work in progress...
  - https://www.mpi-forum.org/mpi-40/

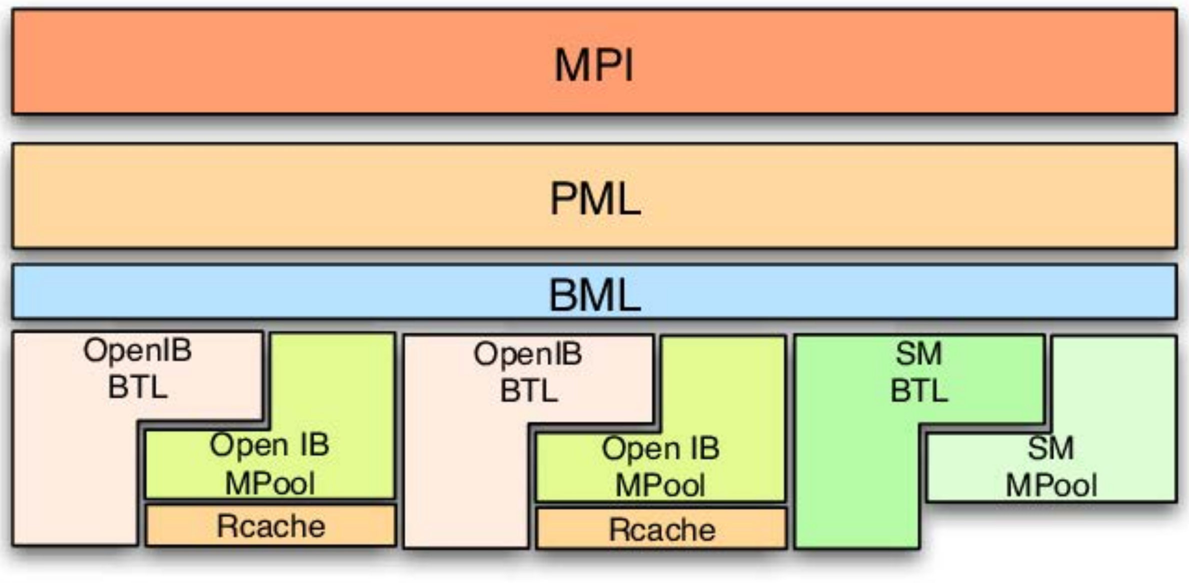
- Some implementations
  - OpenMPI
  - MVAPICH/MPICH
  - Intel MPI
  - IBM BG/Q MPI
  - IBM Spectrum MPI
  - Cray MPI

MVAPICH OPEN MPI intel® IBM CRAY

## OpenMPI stack

- Layered model that abstracts hardware
- Supports various interconnects
- Full support for RDMA
  - ...maps remote memory to local system
- Also supports standard networks like Ethernet

OpenMPI Component Stack



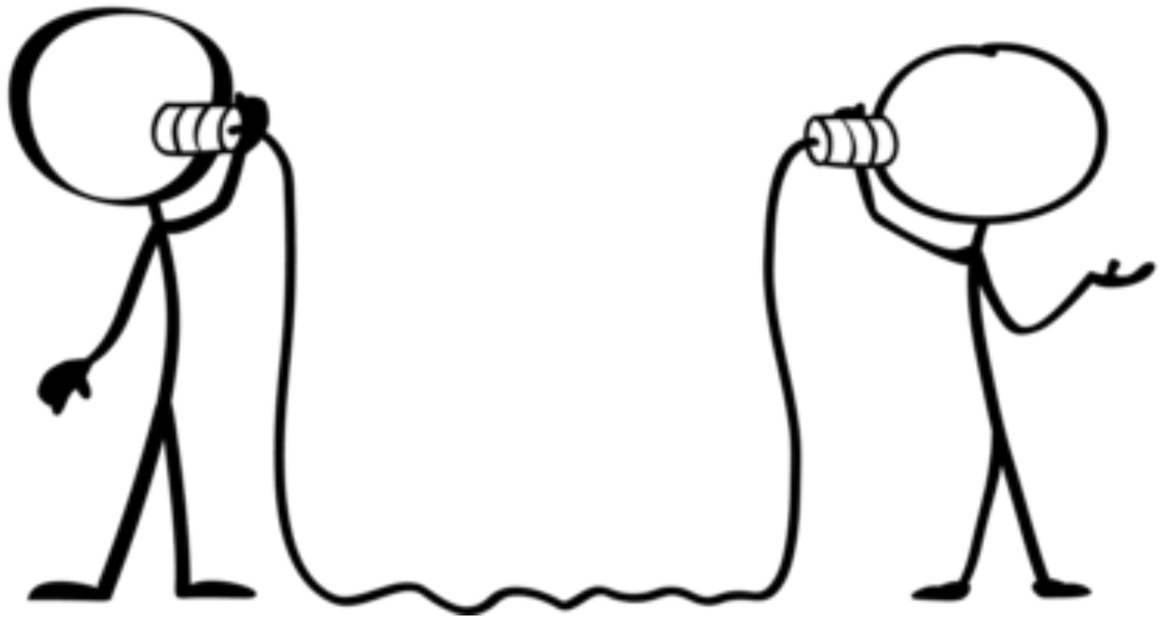
https://open-mpi.org

## Terminology

- **Groups**—multiple processes form a group
- **Context**—a message sent in a context must be received in that context
- **Communicator**—the combination of a group and a context
- **MPI_COMM_WORD**—the default Communicator that talks to all processes
- **Rank**—one instance in an MPI process

## MPI: Point-to-point operations

- **Send/Recv**: send & receive from specific rank to specific rank
  - Several variants:
    - isend/irecv: non-blocking
    - ssend/srecv: syncrhonous mode
- **Sendrecv**: send/recv, but both ends use the same function call
- Common use case:
  - Send information to immediate neighbors in grid



https://www.weshigbee.com/direct-communication-isnt-what-it-seems/

## MPI: Collective operations

- Collective operations are some of the most useful parts of MPI
- Allow for coordination of a large number of ranks trivially
- Common uses:
  - Collect divide-and-conquer pieces of data into central location
  - Distribute work to workers
  - Update global state information

- **Scatter** – split a list and send a piece to each rank
- **Gather** – collect data from all ranks to one rank
- **Allgather** – collect data from all ranks to all ranks
- **Bcast** – send data from one rank to all ranks
- **Reduce** – reduce many values to one with a basic operation
- **Allreduce** – reduce many values to one with a basic operation, send to all ranks
- ...

## Questions?