Plan

Last time

- Domain decomposition for matrix-matrix multiplication
- All nodes shortest path by Floyd-Warshall
- Domain decomposition for traingular matrix-vector multiplication
- Domain decomposition for triangular solve
- Model problem: Gaussian elimination

Today

- Intro to Pthreads
  - examples are taken from `https://computing.llnl.gov/tutorials/pthreads/`

Threads model

A program is a collection of execution paths.

A process is s composed of a sequence of instructions (its code), as well as input and output sets (its data).

- OS maintains information about process such as the program counter, registers, code segment, data segment, stack, heap, etc.

A **threads** is a sequence of instructions within a process that can be scheduled to run by the OS.

- Threads are within the same process address space, which they read and write asynchronously.

Threads model

Threads maintain some private information such as

- pointers to individual stack to store private to the thread values
- private registers
- heap space, shared between threads belonging to the same process

The process itself is the main thread

- it first performs some serial work, and
- then creates a number of threads that are run by the OS.

Threads **communicate** implicitly by writing and reading shared variables.

Locks assure that only one thread at a time updates the same global address within the shared memory.

NOTE: In what follows pthread codes (slightly modifeied) are adopted from `https://computing.llnl.gov/tutorials/pthreads/`

Basic Pthreads types

`pthread_t`	a descriptor and ID
`pthread_mutex_t`	a lock for pthreads
`pthread_cond_t`	a conditional variable
`pthread_attr_t`	a descriptor for pthreads properties
`pthread_mutexattr_t`	a descriptor for mutex properties

Creation and destructions	
`pthread_create`	creates a new thread
`pthread_join`	waits for another thread to return
`pthread_exit`	terminates the calling thread
`pthread_self`	gets a thread's own ID

Creating and destroying locks	
`pthread_mutex_lock`	locking
`pthread_mutex_unlock`	unlocking
`pthread_mutex_trylock`	check lock

Compiling

POSIX threads

Include the header file `#include<pthread.h>`

`gcc fn.c -o fn -lpthread` (other flags as needed)

Creating a thread

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*thread_fun)(void *), void *data);
```

- `pthread_create` creates a new thread of control. It starts execution by invoking function `thread_fun`.
- `pthread_t` is a type (a handle) that gives means for referencing threads. A `pthread_t` variable must exist for every thread created.
- `thread_fun` function executed by `thread`. It must return `void *` and take a single `void *` argument.
- `attr` attributes of `thread` of type `pthread_attr`, default to `NULL`
- `data` argument for `thread_fun` It is passed by reference as a pointer cast of type `void`. If no argument are passed `NULL` is used.
- thread terminates by calling `pthread_exi(tNULL)`

Typical use

```
err = pthread_create(&thread_id, NULL
                              thread_fun, fun_arg);
```

`err` is zero if success and nonzero if not

Creating threads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS   8             /* eight threads will be created */

void *PrintHello(void *thr_id)      /* function executed by threads  */
{                                   /* can have only one argument    */
   long tid = (long) thr_id;        /* user's thread ID              */
   printf("Hello World from thread %ld!\n", tid);
   pthread_exit(NULL);              /* OS terminates the thread      */
}
int main (int argc, char *argv[]) {
   pthread_t thrs[NUM_THREADS];   /* thread handles are defined     */
   int err;
   long t;
   for(t=0; t<NUM_THREADS; t++){
     printf("In main: creating thread %ld\n", t);
     err = pthread_create(&thrs[t], NULL, PrintHello, (void *)t);
   }                                 /* no error if err = 0            */
   pthread_exit(NULL);
}
```

Creating threads

```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #0!
In main: creating thread 3
Hello World! It's me, thread #2!
Hello World! It's me, thread #1!
In main: creating thread 4
Hello World! It's me, thread #3!
In main: creating thread 5
Hello World! It's me, thread #4!
In main: creating thread 6
In main: creating thread 7
Hello World! It's me, thread #6!
Hello World! It's me, thread #5!
Hello World! It's me, thread #7!
```

Passing an entry from global array

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS  3

char *text[NUM_THREADS];      /* defined in global shared memory */
int taskids[NUM_THREADS];

void *PrintHello(void *thr_id) {
   int task_id = *((int*) thr_id);
   printf("Thread %d: %s\n", task_id, text[task_id]);
   pthread_exit(NULL);
}
```

Passing an entry from global array

```
int main(int argc, char *argv[]) {
pthread_t threads[NUM_THREADS];
int rc, t;

text[0] = "English"; text[1] = "French"; text[2] = "Spanish";

for(t=0; t<NUM_THREADS; t++) {
  printf("Creating thread %d\n", t);
  taskids[t] = t;
  rc = pthread_create(&threads[t], NULL, PrintHello,
                      (void *) &taskids[t]);
  }
pthread_exit(NULL);
}
```

Passing an entry from global array

```
Creating thread 0
Creating thread 1
Thread 0: English
Creating thread 2
Thread 1: French
Thread 2: Spanish
```

Passing arguments

`pthread_create()` permits one argument.

When multiple arguments are needed, a pointer to a structure is passed.

The structure is passed by reference and cast to `(void *)`

Passing multiple arguments in structure

```
#include ....
#define NUM_THREADS 3

char *text[NUM_THREADS];
struct thread_data {
   int  thr_id, sum;
   char *text;
};
struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *thr_arg) {
   int taskid, sum;
   char *hello_msg;
   struct thread_data *my_data;
   my_data = (struct thread_data *) thr_arg;
   taskid    = (*my_data).thr_id;   /* same as my_data->the_id;*/
   sum       =  (*my_data).sum;
   hello_msg = (*my_data).text;
   printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
   pthread_exit(NULL);
}
```

Passing multiple arguments in structure

```
int main(int argc, char *argv[]) {
pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t, sum;

sum = 0;
text[0] = "English"; text[1] = "French"; text[2] = "Spanish";

for(t=0;t<NUM_THREADS;t++) {
  sum = t*t;                          /* some random calculation */
  thread_data_array[t].thr_id = t;
  thread_data_array[t].sum    = sum;
  thread_data_array[t].text   = text[t];
  printf("Creating thread %d\n", t);
  pthread_create(&threads[t], NULL, PrintHello, (void *)
       &thread_data_array[t]);
  }
pthread_exit(NULL);
}
```

Passing multiple arguments in structure

```
Creating thread 0
Creating thread 1
Thread 0: English!  Sum=0
Creating thread 2
Thread 1: French!  Sum=1
Creating thread 3
Thread 2: Spanisho  Sum=3
Thread 3: German!  Sum=6
```

Joining threads

Joining threads is one way to accomplish synchronization between threads.

`pthread_join(thr_id,status)` blocks the calling thread until thread `thr_id` terminates.

The target thread's `status` is specified in the target thread's call to `pthread_exit()`.

A joining thread can match exactly one `pthread_join()` call.

Joining threads

Steps:

- Declare a variable `attr` of `pthread_attr_t` type
- Initialize `attr` with `pthread_attr_init(&attr)`
- Set the status with `pthread_attr_setdetachstate()`, `PTHREAD_CREATE_JOINABLE` (The other option is `PTHREAD_CREATE_DETACHED`).
- When done, free the attribute with `pthread_attr_destroy()`

Joining threads

```
#include .....
#define NUM_THREADS 4

void *SomeWork(void *thr_id) {
   int i;                          /* loop work */
   long tid = (long) thr_id;       /* thread ID */
   double sum = 0.0;
   printf("Thread %ld starting...\n",tid);
   srand48(tid);              /* set seed to tid */
   for (i=0; i<10000; i++) {
      sum = sum + drand48()/RAND_MAX;
   }
   printf("Thread %ld done. Result = %e\n",tid, sum);
   pthread_exit((void*) thr_id);
}
```

Joining threads

```
int main (int argc, char *argv[]) {
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc; long t;
   void *status;

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_create(&thread[t], &attr, SomeWork, (void *)t);
      }
   pthread_attr_destroy(&attr);      /* free attribute and wait for others */
   for(t=0; t<NUM_THREADS; t++) {
      rc = pthread_join(thread[t], &status);
      printf("Main completed join with thread %ld, status =
             %ld\n",t,(long)status);
      }
printf("Main thread: program completed. Exiting.\n");
pthread_exit(NULL);
}
```

A `mutex` is a lock protecting access to shared data in order to prevent **race conditions**. Procedure:

- Create and initialize a mutex variable
- Several threads attempt to lock the mutex
- Only one of several threads succeeds in locking the mutex
- The owner thread performs some set of actions on locked data
- The owner unlocks the mutex
- Another thread acquires the mutex and repeats the process
- When all are done, the mutex is destroyed

(1) Lock must be declared and initialized

```
pthread_mutex_t mutexsum;
pthread_mutex_init(&mutexsum, NULL);                    /* dynamic */
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;  /* static */
```

(2) Acquire a lock on the specified mutex variable `mutex`

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

If `mutex` is already locked the thread blocks until `mutex` is unlocked

(3) Unlock a mutex by the owning thread

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Error occures if the mutex was already unlocked

(4) Free a mutex object which is no longer needed

```
pthread_mutex_destroy()
```

Pthreads - dot product

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

#define NUM_THRDS 4
#define VEC_LEN 2000
#define MAX_LEN NUM_THRDS*VEC_LEN

typedef struct {
   double *a, *b;
   double sum;
   int     veclen;
 } dot_data;

dot_data dotstr;
pthread_t thr_id[NUM_THRDS];
pthread_mutex_t mutexsum;
double A[MAX_LEN], B[MAX_LEN];
```

Pthreads - dot product

```
void *dotprod(void *arg) {
   int i, start, end, len ;
   long offset = (long) arg;
   double *x, *y, mysum = 0.0;
   len = dotstr.veclen;
   start = offset*len;
   end   = start + len -1;
   x = dotstr.a;
   y = dotstr.b;
   for (i=start; i<end ; i++) {
      mysum += (x[i] * y[i]);
    }
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);

   pthread_exit(NULL);
}
```

Pthreads - dot product

```
int main (int argc, char *argv[]) {
struct timespec start,finish;
int rc, ntime, stime;
long i,j;
void *status;
pthread_attr_t attr;

for (i=0; i<MAX_LEN; i++) {
  A[i]=1.1/((double) (i+2));         /* random data */
  B[i]= (double) rand()/100000.0;    /* random data */
}

dotstr.veclen = VEC_LEN;
dotstr.a = A;
dotstr.b = B;
dotstr.sum = 0.0;

pthread_mutex_init(&mutexsum, NULL);
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

Pthreads - dot product

```
clock_gettime(CLOCK_REALTIME,&start);
for(i=0;i<NUM_THRDS;i++) {
   rc = pthread_create(&thr_id[i], &attr, dotprod, (void *)i);
}
pthread_attr_destroy(&attr);

for(i=0;i<NUM_THRDS;i++) {
  pthread_join(thr_id[i], &status);
  }

clock_gettime(CLOCK_REALTIME,&finish);
ntime = finish.tv_nsec - start.tv_nsec;
stime = (int) finish.tv_sec - (int) start.tv_sec;
printf("main(): Created %ld threads. Time %ld, nsec %ld\n", NUM_THRDS, stime, ntime

printf ("Sum =  %f \n", dotstr.sum);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);
}
```

Other constructs

Mutexes implement synchronization by controlling thread access to data.

Other synchronizations constructs are

Barrier

Semaphore

Condition variables allow threads to synchronize based on the actual value of data.

Barrier - a point where the thread must wait for other threads

- the thread will proceed only when predefined number of threads reach the same barrier in their respective programs.

`pthread_barrier_t` - barrier type

`pthread_barrier_init` - barrier initialization

```
int pthread_barrier_init(pthread_barrier_t *barrier,
      const pthread_barrierattr_t *attr, unsigned int count);
```

- `barrier` - variable used for the barrier
- `attr` - attributes for the barrier, default NULL
- `count` - number of threads which must `pthread_barrier_wait` on this barrier before the threads can proceed further (not specified which ones)..

Once the barrier is created, each thread will call

`pthread_barrier_wait()`

to indicate that it has completed.

```
int pthread_barrier_wait (pthread_barrier_t *barrier);
```

When a thread calls `pthread_barrier_wait()`, it blocks until the number of threads specified initially in

`pthread_barrier_init()`

function have called

`pthread_barrier_wait()`

All these threads unblock at the same time.

The main can also call `pthread_barrier_wait()`.

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

The barrier should be destroyed only when no thread is executing a wait on the barrier.

Barrier - example

```
#include ....

pthread_barrier_t my_barrier;
......
void *thread1() {
  sleep(2);
  printf("Enter integer value for t1:  ");
  scanf("%d",&t1);
  pthread_barrier_wait(&my_barrier);
  printf("\nvalues entered:  %d %d %d %d \n",t1,t2,t3,t4);
}
void *thread2() {
  sleep(3);
  printf("Enter integer value for t1:  ");
  scanf("%d",&t1);
  pthread_barrier_wait(&my_barrier);
  printf("\nvalues entered:  %d %d %d %d \n",t1,t2,t3,t4);
}
```

Barrier example

```
main() {
pthread_t thread_id_1,thread_id_2;
pthread_attr_t attr;
.....
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
pthread_barrier_init(&my_barrier,NULL,2);

pthread_create(&thread_id_1,NULL,&thread1,NULL);
pthread_create(&thread_id_2,NULL,&thread2,NULL);

pthread_join(thread_id_1,NULL);
pthread_join(thread_id_2,NULL);

pthread_barrier_destroy(&my_barrier);
pthread_exit(NULL);
}
```

Barriers are for global synchronization

- common when running multiple copies of the same function (SIMD)
- a thread reaching a barrier stalls until all other participating threads reach the barrier.

Example: Iterative linear system solvers: `x_new = A*x_old + r;`

```
x_new[i*k:i*(k+1)-1] = A[i*k:i*(k+1)-1,:]*x_old;
barrier_wait;
x_old[i*k:i*(k+1)-1] = x_new [i*k:i*(k+1)-1];
barrier_wait;
```

Not all implementations provide `barrier` funcionality.

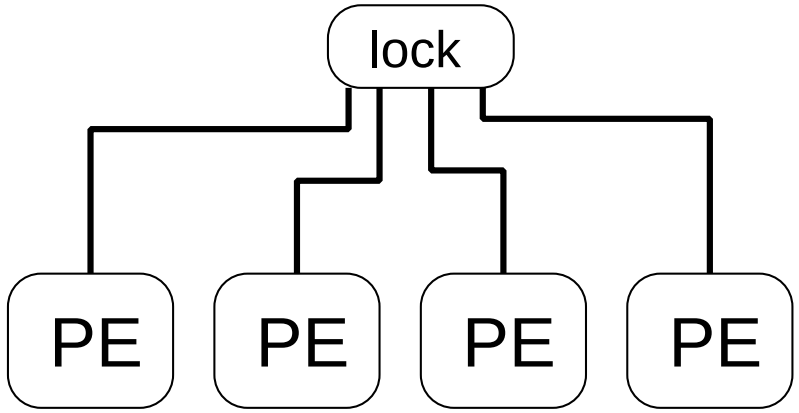
A crude way to implement barrier with locks

```
barrier(int *counter, int thread_count)
{
pthread_mutex_lock(&counter_lock);
*counter++;
pthread_mutex_unlock(&counter_lock);
while (*counter < thread_count) {};
}
```

Use a counter that keeps track of how many threads have reached the barrier.

On reaching the barrier, a thread increments the counter and (busy) waits for the counter to reach the value `thread_count`.

Barrier is a hot spot



Spread hot spots, use a tree