THREAD PROGRAMMING

Explicit Synchronization: Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- The second argument specifies an object attribute; using NULL yields the default attributes.
- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

- This barrier could have been statically initialized by assigning an initial value created using the macro

  `PTHREAD_BARRIER_INITIALIZER(3).`

Slide source: Jim Demmel and Kathy Yelick

Calculating Π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
   sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

Serial code for calculating Π

Parallel Version

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

Accuracy of Parallel and Serial on Dual core

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Why serial is more accurate?

Because the same variable *sum* is being updated in parallel!

One Solution: Busy waiting with turn flag

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

Mutexes (aka Locks) in Pthreads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

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

- Multiple mutexes may be held, but can lead to deadlock:

```
thread1              thread2
lock(a)              lock(b)
lock(b)              lock(a)
```

Slide source: Jim Demmel and Kathy Yelick

Another Solution: Using Mutex

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

Time Comparison

Table 4.1 Run-Times (in Seconds) of π Programs Using $n = 10^8$ Terms on a System with Two Four-Core Processors

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

Conditional Wait/Signal

- Block the thread on a conditional variable
- The thread will wake up when a signal is raised.

int pthread_cond_init(pthread_cond_t **cond*, const pthread_condattr_t **attr*);

int pthread_cond_wait(pthread_cond_t **cond*, pthread_mutex_t **mutex*);

int pthread_cond_signal(pthread_cond_t *cond);

Shared Memory

- Dynamic threads
  - Master thread waits for work, forks new threads, and when threads are done, they terminate
  - Efficient use of resources, but thread creation and termination is time consuming.
- Static threads
  - Pool of threads created and are allocated work, but do not terminate until cleanup.
  - Better performance, but potential waste of system resources.
  - Next page example:
    - A static thread pool to execute simple calculation works

Example – Using Thread Pool

```
#include "queue.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define THREADS 3
/** Task queue. */
QUEUE queue;
/** Type of a calc work task. */
typedef struct {
    int a;
    int b;
    int type;
    QUEUE node;
} work_t;
```

Definitions

```
/** Our threads.*/
pthread_t threads[THREADS];
/**Our thread condition variable.*/
pthread_cond_t cond;
/**Our thread mutex lock.*/
pthread_mutex_t mutex;
/* function headers */
void * worker();
void submit_work(int a, int b, int type);
/** Should execute the submited work tasks through thread pool. */
```

```
int main(void) {
    QUEUE_INIT(&queue);
    pthread_cond_init(&cond, NULL);
    pthread_mutex_init(&mutex, NULL);
    /* 3 + 3 = 6 */
    submit_work(3, 3, 1);
    /* 4 - 3 = 1 */
    submit_work(4, 3, 2);
    /* 7 * 8 = 56 */
    submit_work(7, 8, 3);
    /* 30 / 6 = 5 */
    submit_work(30, 6, 4);
```

Starting threads

```
/* start all threads */
   for (int i = 0; i < THREADS; i++)
      pthread_create(&threads[i], NULL, worker, NULL);
   /* wait all threads to finish */
   for (int i = 0; i < THREADS; i++)
      pthread_join(threads[i], NULL);
   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&cond);
   return EXIT_SUCCESS;
}
```

Work submission

```
void submit_work(int a, int b, int type) {
    work_t * work = malloc(sizeof(work_t));
    work->a = a;
    work->b = b;
    work->type = type;
     pthread_mutex_lock(&mutex);
    QUEUE_INIT(&work->node);
    QUEUE_INSERT_TAIL(&queue, &work->node);
    pthread_mutex_unlock(&mutex);
    /* signal a thread that it should check for new work */
    pthread_cond_signal(&cond);
}
```

Worker thread. Looks for new tasks to execute

```
void * worker() {
    QUEUE * q;
    int result;
    bool spin = true;
    work_t * work;
    while (spin) {
        pthread_mutex_lock(&mutex);
        while (QUEUE_EMPTY(&queue)) {
            pthread_cond_wait(&cond, &mutex);
        }
        q = QUEUE_HEAD(&queue);
        QUEUE_REMOVE(q);
        pthread_mutex_unlock(&mutex);
         work = QUEUE_DATA(q, work_t, node);
```

```
    switch (work->type) {
        case 1:
            result = work->a + work->b; break;
        case 2:
            result = work->a - work->b; break;
        case 3:
            result = work->a * work->b; break;
        case 4:
            result = work->a / work->b; break;
        default: spin = false;
    }
    free(work);
  }//while(spin)
  pthread_exit(NULL);
}
```

Thread Safety

- Chapter 2 mentions thread safety of shared-memory parallel functions or libraries.
  - A function or library is thread-safe if it operates “correctly” when called by multiple, simultaneously executing threads.
  - Since multiple threads communicate and coordinate through shared memory, a thread-safe code modifies the state of shared memory using appropriate synchronization.
  - Some features of sequential code that may not be thread safe?

Summary of Programming with Threads

- Pthreads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most programmers
  - Ability to shared data is convenient
- Pitfalls
  - Data races are difficult to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- **OpenMP** is commonly used today as a simpler alternative, but it is more restrictive
  - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence

OPENMP PROGRAMMING

OpenMP: Prevailing Shared Memory Programming Approach

- Model for shared-memory parallel programming
- Portable across shared-memory architectures
- Scalable (on shared-memory platforms)
- Incremental parallelization
  - Parallelize individual computations in a program while leaving the rest of the program sequential
- Compiler based
  - Compiler generates thread program and synchronization
- Extensions to existing programming languages (Fortran, C and C++)
  - mainly by directives
  - a few library routines

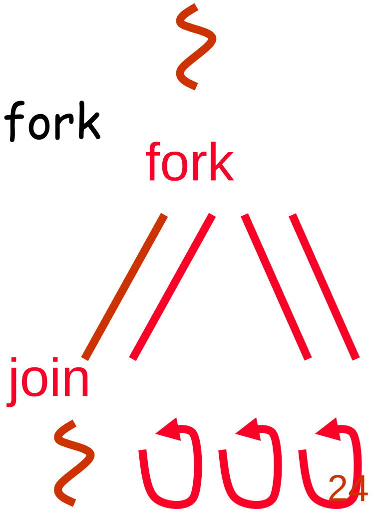
See http://www.openmp.org

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

OpenMP Execution Model

- Fork-join model of parallel execution
- Begin execution as a single process (**master thread**)
- Start of a parallel construct:
  - Master thread creates team of threads (**worker threads**)
- Completion of a parallel construct:
  - Threads in the team synchronize -- **implicit barrier**
- Only master thread continues execution
- Implementation optimization:
  - Worker threads spin waiting on next fork



OpenMP uses Pragmas

- Pragmas are special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C specification.
- Compilers that don't support the pragmas ignore them.
- The interpretation of OpenMP pragmas
  - They modify the statement immediately following the pragma
  - This could be a compound statement such as a loop

```
#pragma omp …
```

Programming Model – Data Sharing

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)
- PThreads:
  - Global-scoped variables are shared
  - Stack-allocated variables are private
- OpenMP:
  - `shared` variables are shared
  - `private` variables are private
  - Default is `shared`
  - Loop index is `private`

```
// shared, globals
int bigdata[1024];

void* foo(void* bar) {
  int tid;

  #pragma omp parallel \
   shared ( bigdata ) \
   private ( tid )
  {
    /* Calc. here */
  }
}
```

In case the compiler doesn't support OpenMP

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

```
#ifdef _OPENMP
# include <omp.h>
#endif
```

OpenMP directive format C (also Fortran and C++ bindings)

- Pragmas, format

  `#pragma omp` directive_name [clause [clause] ...] new-line

- Conditional compilation

```
#ifdef _OPENMP
    block,
    e.g., printf("%d avail.processors\n",omp_get_num_procs());
#endif
```

- Case sensitive
- Include file for library routines

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

OpenMP runtime library, Query Functions

`omp_get_num_threads`:

Returns the number of threads currently in the team executing the parallel region from which it is called

```
int omp_get_num_threads(void);
```

`omp_get_thread_num`:

Returns the thread number, within the team, that lies between `0` and `omp_get_num_threads()-1`, inclusive. The master thread of the team is thread `0`

```
int omp_get_thread_num(void);
```

OpenMP parallel region construct

- Block of code to be executed by multiple threads in parallel
- Each thread executes the **same code redundantly (SPMD)**
  - Work within work-sharing constructs is distributed among the threads in a team
- Example with C/C++ syntax

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

structured-block

- clause can include the following:

`private` (list)

`shared` (list)

Hello World in OpenMP

- Let's start with a parallel region construct
- Things to think about
  - As before, number of threads is read from command line
  - Code should be correct without the pragmas and library calls
- Differences from Pthreads
  - More of the required code is managed by the compiler and runtime (so shorter)
  - There is an implicit thread identifier

gcc –fopenmp …

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

In case the compiler doesn't support OpenMP

```
# ifdef _OPENMP
   int my_rank = omp_get_thread_num ( );
   int thread_count = omp_get_num_threads ( );
# e l s e
   int my_rank = 0;
   int thread_count = 1;
# endif
```

OpenMP Data Parallel Construct: Parallel Loop

- All pragmas begin: #pragma
- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning of Res
- Synchronization also automatic (barrier)

Serial Program:

```
void main()
{
    double Res[1000];

    for(int i=0;i<1000;i++) {
            do_huge_comp(Res[i]);
    }
}
```

Parallel Program:

```
void main()
{
    double Res[1000];
#pragma omp parallel for
    for(int i=0;i<1000;i++) {
            do_huge_comp(Res[i]);
    }
}
```

Limitations and Semantics

- Not all "element-wise" loops can be parallelized

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

  - Loop index: signed integer
  - Termination Test: <,<=,>,=> with loop invariant int
  - Incr/Decr by loop invariant int; change each iteration
  - Count up for <,<=; count down for >,>=
  - Basic block body: no control in/out except at top

- Threads are created and iterations divvied up; requirements ensure iteration count is predictable

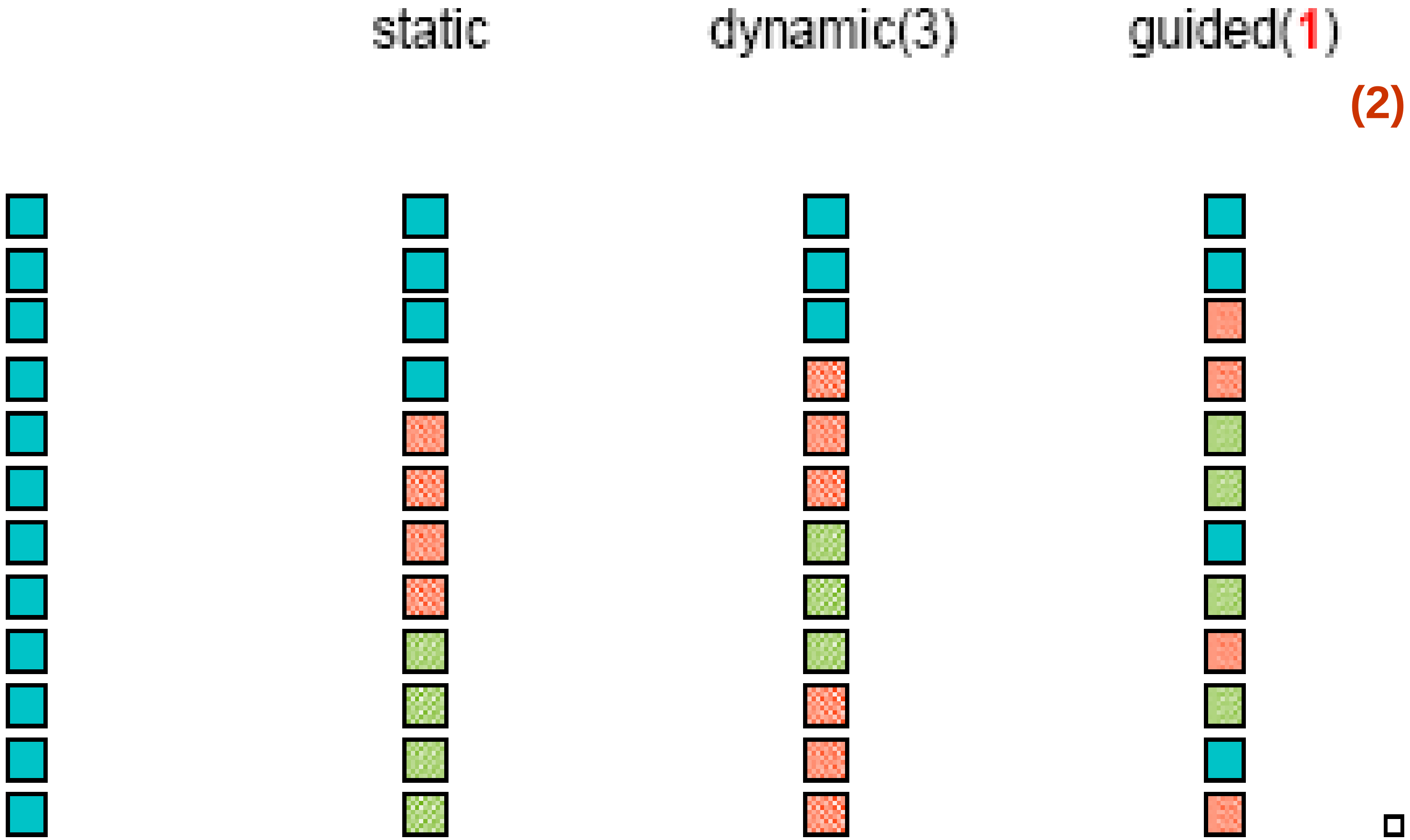
OpenMP Synchronization

- Implicit barrier
  - At beginning and end of parallel constructs
  - At end of all other control constructs
  - Implicit synchronization can be removed with **`nowait`** clause
- Explicit synchronization
  - `critical`
  - `atomic`

Programming Model – Loop Scheduling

- `schedule` clause determines how loop iterations are divided among the thread team
  - **`static([chunk])`** divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default **`[chunk]`** is **`ceil( # iterations / # threads )`**
  - **`dynamic([chunk])`** allocates **`[chunk]`** iterations per thread, allocating an additional **`[chunk]`** iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default **`[chunk]`** is 1
  - **`guided([chunk])`** allocates dynamically, but **`[chunk]`** is exponentially reduced with each allocation

Loop scheduling



More loop scheduling attributes

- RUNTIME The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- AUTO The scheduling decision is delegated to the compiler and/or runtime system.
- **NO WAIT / nowait**: If specified, then threads do not synchronize at the end of the parallel loop.
- **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause (collapsed order corresponds to original sequential order).

Impact of Scheduling Decision

- Load balance
  - Same work in each iteration?
  - Processors working at same speed?
- Scheduling overhead
  - Static decisions are cheap because they require no run-time coordination
  - Dynamic decisions have overhead that is impacted by complexity and frequency of decisions
- Data locality
  - Particularly within cache lines for small chunk sizes
  - Also impacts data reuse on same processor

Summary of Lecture

- OpenMP, data-parallel constructs only
  - Task-parallel constructs later
- What's good?
  - Small changes are required to produce a parallel program from sequential (parallel formulation)
  - Avoid having to express low-level mapping details
  - Portable and scalable, correct on 1 processor
- What is missing?
  - Not completely natural if want to write a parallel code from scratch
  - Not always possible to express certain common parallel constructs
  - Locality management
  - Control of performance