The Pthread API

(see https://computing.llnl.gov/tutorials/pthreads/)

- Pthreads has emerged as the standard threads API (Application Programming Interface), supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.
- Provides two basic functions for specifying concurrency:

```c
#include <pthread.h>

int pthread_create (pthread_t *thread_handle,
                   const pthread_attr_t *attribute,
                   void (*thread_function)(void *),
                   void *arg);

int pthread_join (pthread_t thread_handle,
                  void *ptr);
```

Mutual Exclusion

- Critical sections in Pthreads are implemented using mutex locks.
- Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation.
- A thread entering a critical section first tries to get a lock. It goes ahead when the lock is granted.
- The API provides the following functions for handling mutex-locks:

```c
int pthread_mutex_lock ( pthread_mutex_t *mutex_lock);

int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);

int pthread_mutex_init (pthread_mutex_t *mutex_lock,
                         const pthread_mutexattr_t *lock_attr);
```

Can replace by NULL.
An Example (compute \( \pi \))

The value of PI can be calculated in a number of ways. Consider the following method of approximating PI:

- Inscribe a circle in a square
- Randomly generate points in the square
- Determine the number of points in the square that are also in the circle
- Let \( A_c/A_s \) be the number of points in the circle divided by the number of points in the square
- \( \pi \sim 4 \times (A_c/A_s) \)
- Note that the more points generated, the better the approximation

\[
\begin{align*}
A_c &= (2r)^2 = 4r^2 \\
A_s &= \pi r^2 \\
\pi &= 4 \times \frac{A_c}{A_s}
\end{align*}
\]

#include <sys/time.h>
#define MAX_THREADS 64

void *compute_pi (void *);

int total_hits, sample_points, sample_points_per_thread, num_threads;

main ( ) {
    ...
}

void *compute_pi (void *s) {
    ...
}
An Example (compute $\pi$)

struct arg_to_thread {int t_seed ; int hits ;}

main ( int argc, char argv[] )    {
    sample_points = atoi(argv[1]) ; /* first argument is the number of points */
    num_threads = atoi(argv[2]) ;   /* second argument is the number of threads*/

    int i ;
    pthread_t  p_threads[MAX_THREADS];
    pthread_attr_t  attr;
    double computed_pi;
    struct arg_to_thread my_arg[MAX_THREADS] ;

    pthread_attr_init (&attr);

    total_hits =0;
    sample_points_per_thread = sample_points /num_threads;

    for (i=0; i< num_threads; i++){
        my_arg[i].t_seed = i;   /* can chose any seed – here i is chosen*/
        pthread_create (&p_threads[i], &attr, compute_pi, &my_arg[i]);
    }

    for (i=0; i< num_threads; i++){
        pthread_join (p_threads[i], NULL);
        total_hits += my_arg[i].hits;
    }

    computed_pi = 4.0*(double) total_hits / ((double) (sample_points));
}
An Example (compute $\pi$)

```c
void *compute_pi (void *s) {
    struct arg_to_thread *local_arg;
    int seed, i, local_hits;
    double rand_no_x, rand_no_y;

    local_arg = s;
    seed = (*local_arg).t_seed;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x = (double) (rand_r (&seed))/(double) RAND_MAX ;
        rand_no_y = (double) (rand_r (&seed))/(double) RAND_MAX ;
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++; /* the generated sample is inside the circle*/
        seed *= i;
    }

    (*local_arg).hits = local_hits;
    pthread_exit (0);
}
```

Re-entrant function to generate a random number between 0 and RAND_MAX
Need to compile with “gcc -D_REENTRANT -lpthread”

Allows the removal of “total_hits += my_arg[i].hits;” from main()
Multiprocessors connected by networks (Section 6.7)

- Each processor has private physical address space

Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system (ex: Ethernet or a switch)
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problem: Low interconnect bandwidth (compared to SMP)

- Grid Computing
  - computers interconnected by long-haul networks (ex: Internet)
  - Work units farmed out, results sent back
  - Can make use of idle time on PCs (ex: PITTGRID)
Programming a distributed address space machine

- Assume that 10000 values are stored in the local memories of 16 processors such that 625 values are stored in \( x[0] \ldots x[624] \) in the local memory of each processor.
- All variables are local variables (each processor has its own copy) – no shared variables.
- The function "send(\( m, p \))" sends a message containing the value of \( m \) to processor \( p \).
- The function "receive(\( m \))" receives a message and puts the received value in \( m \).

\[
\begin{align*}
\text{sum} &= 0; \\
\text{for (} i=0 \ ; \ i < 625 \ ; \ i++ \) \\
\text{sum} &= \text{sum}+ x[i]; \\
\text{half} &= 8; \quad \text{\textit{P} = 16 \ \textit{I}/} \\
\text{for (} i=0 \ ; \ i < 4 \ ; \ i++ \) \\
\{ & \text{if (} 2 \times \text{half} > \text{Pid} \geq \text{half} \} \ \text{send (sum, Pid - half);} \\
& \text{if (} \text{Pid} < \text{half} \} \ \{ \ \text{receive (remote_sum);} \ ; \\
& \text{sum} += \text{remote_sum} ; \}
\end{align*}
\]

- No shared variables.
- Where is the global sum?
- The distribution of the initial data to the local memories is done either by the programmer or by the compiler.

Interconnection network (Section 6.8)

- To connect processors to memories or processors to processors

- Issues
  - Latency
  - Bandwidth
  - Cost (wires, switches, ports, …)
  - Scalability

- Topology has been a focus of architects
Evaluating Interconnection Network topologies

• **Diameter**: The distance between the farthest two nodes in the network.

• **Average distance**: The average distance between any two nodes in the network.

• **Node degree**: The number of neighbors connected to any particular node.

• **Bisection Width**: The minimum number of wires you must cut to divide the network into two equal parts.

• **Cost**: The number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost. However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor in to the cost.

---

**Buses and crossbars**

- **Cost**
- **Latency**
- **Bandwidth**
- **Scalability**

Each switch is a 2x2 switch that can be set to one of 2 settings.
Multistage networks

- **NxN Omega network**: log N stages, with N/2, 2x2 switches.
- **A blocking network**: some input-output permutations cannot be realized due to path conflicts.

Circuit switching: circuits are established between inputs and outputs – arbitrate entire circuits.

Packet switching: packets are buffered at intermediate switches – arbitrate individual switches.

2-D torus

- Diameter??
- Bisection bandwidth??
- Routing algorithms
  - x-y routing
  - Adaptive routing
- 2D mesh (without the wrap-around connections)

- **Variants**
  - 1-D (ring), 3-D.
Hypercube interconnections

- An interconnection with low diameter and large bisection width.
- A q-dimensional hypercube is built from two (q-1)-dimensional hypercubes.

1-dimension binary hypercube

2-dimension binary hypercube

3-dimension binary hypercube

A 4-dimension Hypercube (16 nodes)

- Can recursively build a q-dimension network – has $2^q$ nodes
A fat tree networks using 2x2 bidirectional switches