Programming Shared Address Space Platforms -- POSIX and OpenMP

Ananth Grama
Computing Research Institute and
Department of Computer Sciences,
Purdue University.

`ayg@cs.purdue.edu`
`http://www.cs.purdue.edu/people/ayg`

Reference:

Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, Vipin Kumar, George Karypis, Addison Wesley, ISBN: 0-201-64865-2, 2003.

Thread Basics

A thread is a single stream of control in the flow of a program.

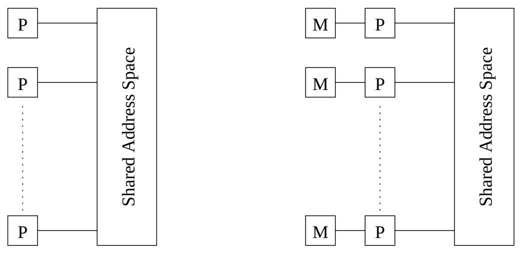
A simple code fragment such as:

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

can be threaded as:

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(
               dot_product(get_row(a, row),
                        get_col(b, col)));
```

Logical Memory Model of a Thread



All memory is global and equally accessible to all threads. However, in practice, it is desirable to treat stack resident data as local to threads.

Why Threads?

Software Portability

Latency Hiding

Scheduling and Load Balancing

Serial Performance

Ease of Programming, Widespread Use

The POSIX Thread API

Thread Basics: Creation and Termination

```
#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,
    void   **ptr);
```

Thread Basics: Creation and Termination

```
main() {
    int i;
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t  attr;
    pthread_attr_init (&attr);

    // initializations..

    for (i=0; i< num_threads; i++) {
        hits[i] = i;
       pthread_create(&p_threads[i], &attr,
            compute_pi, (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
}

void *compute_pi (void *s) {
    // function here
}
```

Synchronization Primitives in Pthreads

Mutual Exclusion for Shared Variables

Consider the following statement executed by all threads:

```
if (my_cost < best_cost)
    best_cost = my_cost;
```

The result is clearly non-deterministic.

To support such segments, POSIX supports mutual exclusion primitives:

```
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);
```

Mutual Exclusion

```
#include <pthread.h>
void *find_min(void *list_ptr);
pthread_mutex_t minimum_value_lock;
int minimum_value, partial_list_size;

main() {
    // declare and initialize data structures and list
    minimum_value = MIN_INT;
    pthread_init();
    pthread_mutex_init(&minimum_value_lock, NULL);

   // initialize lists, list_ptr, and partial_list_size
    // create and join threads here
}

void *find_min(void *list_ptr) {
    int *partial_list_pointer, my_min, i;
    // more initializations

    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
        minimum_value = my_min;
    // and unlock the mutex
    pthread_mutex_unlock(&minimum_value_lock);
    pthread_exit(0);
}
```

Notes on Mutexes:

Mutexes are serialization constructs. For this reason, these segments must be made as small as possible. Other constructs such as trylocks and read-write locks can improve performance.

Condition Variables for Synchronization

Interrupt-based mechanism for synchronizing threads.

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


int pthread_cond_signal(pthread_cond_t
*cond);

int pthread_cond_broadcast(pthread_cond_t
*cond);


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

Composite Synchronization Constructs

A number of more complex constructs, such as barriers and read-write locks can be efficiently implemented using mutexes and condition variables (see notes).

The OpenMP Programming Model

Uses the thread model to support directive-based parallelism.

The default OpenMP directive is as follows:

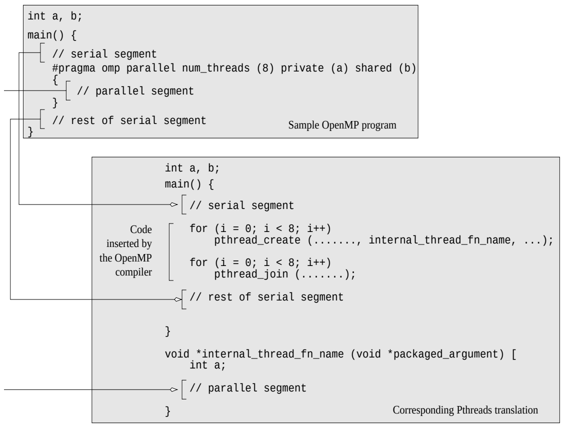
```
#pragma omp directive [clause list]
```

OpenMP programs execute serially until they encounter the parallel directive.

```
#pragma omp parallel [clause list]
/* structured block */
```

Each thread created by this directive executes the structured block specified by the parallel directive. The clause list is used to specify conditional parallelization, number of threads, and data handling.

The OpenMP Programming Model


Sample OpenMP program
Code inserted by the OpenMP compiler
Corresponding Pthreads translation

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

```
int a, b;
main() {
    // serial segment

    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);

    // rest of serial segment

}
void *internal_thread_fn_name (void *packaged_argument) [
    int a;

    // parallel segment

}
```

An OpenMP program with its POSIX translation.

Using the `parallel` directive

```
#pragma omp parallel if (is_parallel == 1) \
                          num_threads(8) \
                          private (a) \
                          shared (b) \
                          firstprivate(c)
{
    /* structured block */
}
```

Here, if the value of the variable isparallel equals one, eight threads are created. Each of these threads gets private copies of variables a and c, and shares a single value of variable b. Furthermore, the value of each copy of c is initialized to the value of c before the parallel directive.

Specifying Concurrent Tasks in OpenMP

The parallel directive can be used in conjunction with other directives to specify concurrency across iterations and tasks. OpenMP provides two directives -- for and sections -- to specify concurrent iterations and tasks.

The for Directive:

```
    #pragma omp for [clause list]
        /* for loop */
```

The clauses that can be used in this context are: private, firstprivate, lastprivate, reduction, schedule, nowait, and ordered.

The sections Directive:

```
#pragma omp sections [clause list]
{
    [#pragma omp section
        /* structured block */
    ]
    [#pragma omp section
        /* structured block */
    ]
    ...
}
```

Synchronization Constructs in OpenMP

```
#pragma omp barrier

#pragma omp single [clause list]
         structured block

#pragma omp master
         structured block

#pragma omp critical [(name)]
         structured block

#pragma omp ordered
    structured block
```

OpenMP Library Functions

Controlling Number of Threads and Processors

```
#include <omp.h>

void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

Controlling and Monitoring Thread Creation

```
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
```

Mutual Exclusion

```
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

Environment Variables in OpenMP

```
OMP_NUM_THREADS
OMP_SET_DYNAMIC
OMP_DYNAMIC
OMP_NESTED
OMP_SCHEDULE

setenv OMP_SCHEDULE "dynamic"
setenv OMP_SCHEDULE "guided"
```