Programming Shared Address Space Platforms (Chapter 7)

Alexandre David

B2-206

Today

- Thread Basics (7.1 – 7.4).
- Synchronization Primitives in Pthreads (7.5).

Comparison

- Explicit parallel programming: specify tasks and interactions.
  - Communication of intermediate results.
  - Synchronization.
- MPI: focus on communication.
- Shared memory: focus on synchronization.

Programming Models

- Concurrency supported by:
  - Processes – private data unless otherwise specified.
  - Threads – shared memory, lightweight.
  - Directive based programming – concurrency specified as high level compiler directive, OpenMP.
- See OS course.

Threads Basics

- All memory is globally accessible.
- But the stack is considered local.
  - In practice both local (private) and global (shared) memory.
  - Recall that memory is physically distributed and local accesses are faster.

Why Threads?

- Software portability – applications developed and run without modification on multi-processor machines.
- Latency hiding – recall chapter 2.
- Implicit scheduling and load balancing – specify many tasks and let the system map and schedule them.
- Ease of programming, widespread.

The POSIX Thread API

- It is a standard API (like MPI).
  - Supported by most vendors.
- General concepts applicable to other thread APIs (java threads, NT threads, etc).
- Low level functions, API is missing high level constructs, e.g., no collective communication like in MPI.

Thread Creation

```
#include <pthread.h>

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

Header.

Identifier.

NULL for default.

Function to call with its argument.

Waiting for Termination

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

Thread to wait for.

Threads call pthread_exit(value).
The caller can read a (void*) at address *ptr*.

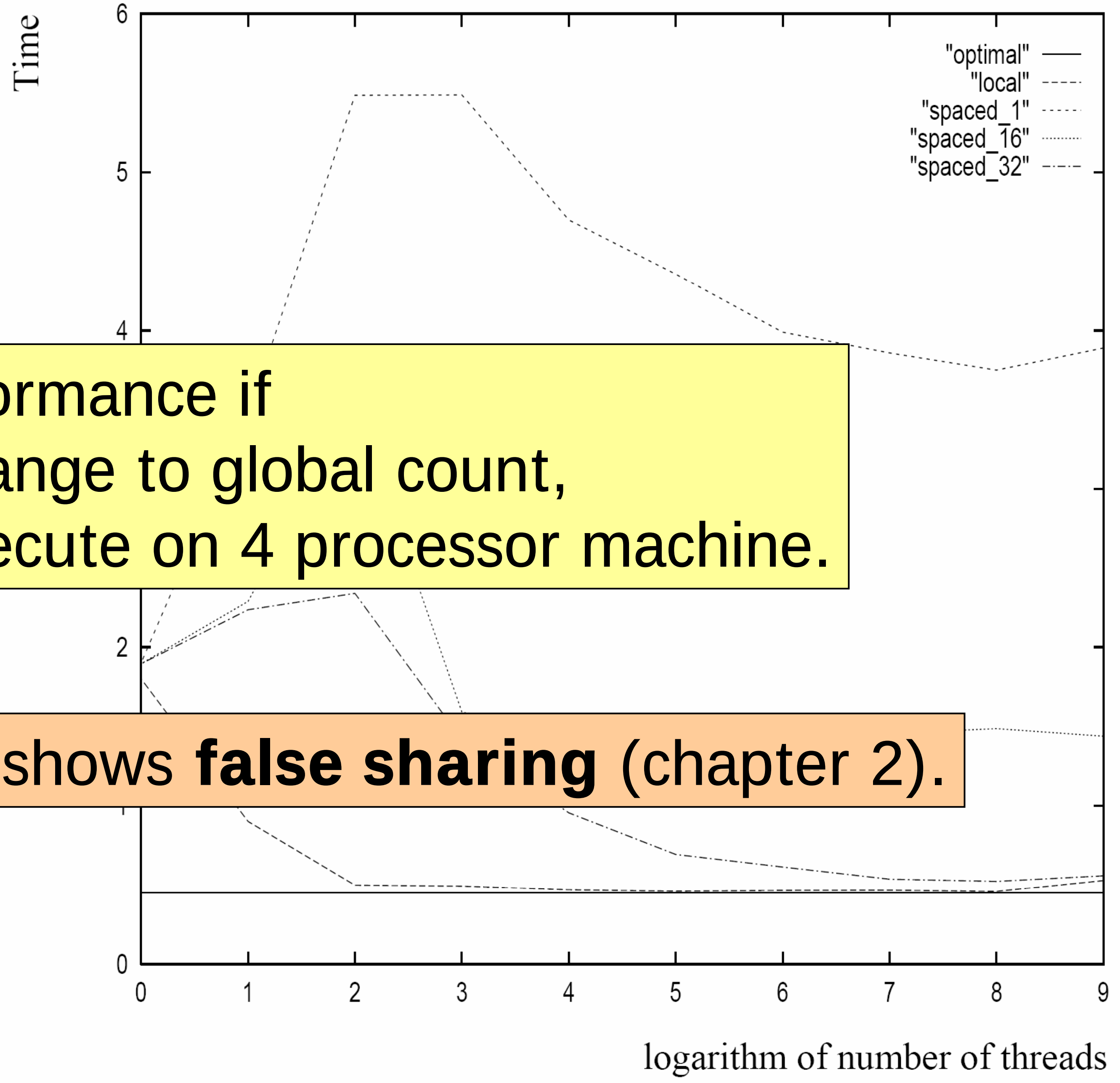
The creator process/thread calls this function to wait for its spawned threads.

Example: Compute PI

```
#include <pthread.h>
...
main() {
    ...
        pthread_t p_threads[MAX_THREADS];
        pthread_attr_t attr;
        pthread_attr_init (&attr);
        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];
        }
```

Example: Compute PI

```
void *compute_pi (void *s) {
        int seed, i;
        double rand_no_x, rand_no_y;
        int *hit_pointer = (int *) s;                    To return the result.
        seed = *hit_pointer;                             Used to pass seed.
        int local_hits = 0;
        for (i = 0; i < sample_points_per_thread; i++) {
                rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
                rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
                if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
Count hits          (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
in the circle.              local_hits ++;
                seed *= i;
        }
        *hit_pointer = local_hits;                       Return result.
        pthread_exit(0);
}
```



Performance if

- change to global count,
- execute on 4 processor machine.

This shows **false sharing** (chapter 2).

Figure 7.2 Execution time of the `compute_pi` program as a function of number of threads.

Race Condition

- Need to synchronize if a shared variable is updated concurrently.
  - `if (my_cost < best_cost) best_cost = my_cost;`
  - Race condition.
  - Can give wrong (inconsistent) result.
  - We want this to be atomic – but we can't so this is a critical segment: Must be executed by only one thread at a time.

Mutex-Locks

- Implement critical section.
- Mutex-locks can be locked or unlocked.
  - Locking is atomic.
  - Threads must acquire a lock to enter a critical section.
  - Threads must release their locks when leaving a critical section.
- Locks represent serialization points. Too many locks will decrease performance.

Mutex-Lock

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

int pthread_mutex_lock(
        pthread_mutex_t *mutex_lock);

int pthread_mutex_unlock(
        pthread_mutex_t *mutex_lock);
```

Example Revisited

```
pthread_mutex_t minimum_value_lock;
...
main() {
	...
		pthread_mutex_init(&minimum_value_lock, NULL);
	....
}
void *find_min(void *list_ptr) {
		...
		pthread_mutex_lock(&minimum_value_lock);
		if (my_min < minimum_value) minimum_value = my_min;
		pthread_mutex_unlock(&minimum_value_lock);
		...
}
```