



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

并行程序设计 with 算法（实验）

3-Pthreads 并行矩阵乘法与数组求和

吴迪、刘学正
中山大学计算机学院

◉ 内容

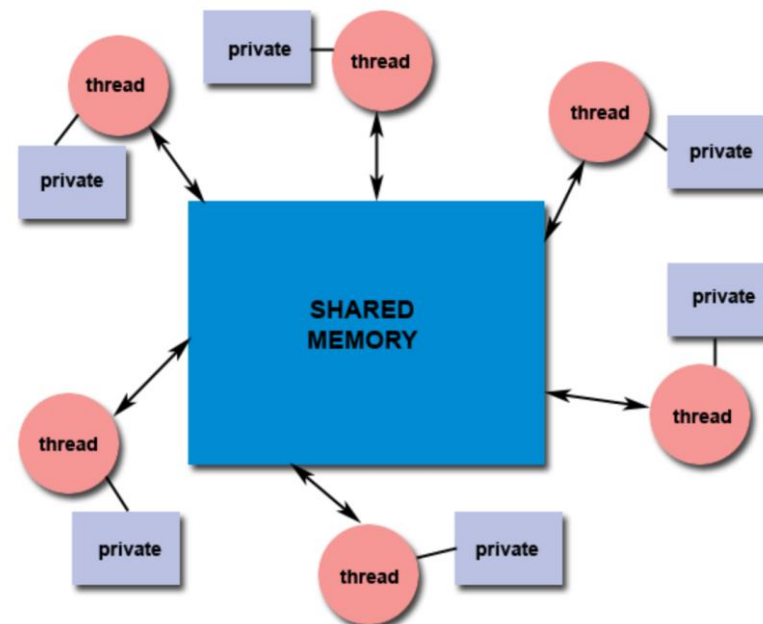
- Pthreads程序编写、运行与调试
- 多线程并行矩阵乘法
- 多线程并行数组求和

◉ 目的

- 掌握Pthreads编程的基本流程
- 理解线程间通信与资源共享机制
- 通过性能分析明确线程数、数据规模与加速比的关系

- 定义：Pthreads是一套POSIX线程标准，用于在UNIX系统上创建和管理线程
- 为什么要用Pthreads？
 - 轻量级：相对于进程，线程创建 `pthread_create()` 开销更低
 - 高效的数据交换：多个线程直接共享内存地址，免数据传输

| Platform | fork() | | | pthread_create() | | |
|--|--------|------|------|------------------|------|-----|
| | real | user | sys | real | user | sys |
| Intel 2.6 GHz Xeon E5-2670 (16 cores/node) | 8.1 | 0.1 | 2.9 | 0.9 | 0.2 | 0.3 |
| Intel 2.8 GHz Xeon 5660 (12 cores/node) | 4.4 | 0.4 | 4.3 | 0.7 | 0.2 | 0.5 |
| AMD 2.3 GHz Opteron (16 cores/node) | 12.5 | 1.0 | 12.5 | 1.2 | 0.2 | 1.3 |
| AMD 2.4 GHz Opteron (8 cores/node) | 17.6 | 2.2 | 15.7 | 1.4 | 0.3 | 1.3 |
| IBM 4.0 GHz POWER6 (8 cpus/node) | 9.5 | 0.6 | 8.8 | 1.6 | 0.1 | 0.4 |
| IBM 1.9 GHz POWER5 p5-575 (8 cpus/node) | 64.2 | 30.7 | 27.6 | 1.7 | 0.6 | 1.1 |
| IBM 1.5 GHz POWER4 (8 cpus/node) | 104.5 | 48.6 | 47.2 | 2.1 | 1.0 | 1.5 |
| INTEL 2.4 GHz Xeon (2 cpus/node) | 54.9 | 1.5 | 20.8 | 1.6 | 0.7 | 0.9 |
| INTEL 1.4 GHz Itanium2 (4 cpus/node) | 54.5 | 1.1 | 22.2 | 2.0 | 1.2 | 0.6 |



- 引入 Pthreads 头文件
 - #include <pthread.h>
- 定义线程函数 (POSIX 标准)
 - void* thread_function(void* arg);
- 存储线程 ID
 - pthread_t 变量
- 创建线程
 - pthread_create()
- 等待线程结束
 - pthread_join()
- 显式终止线程 (可选)
 - pthread_exit()

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int thread_count;
6
7  void* Hello(void* rank);
8
9  int main (int argc, char *argv[])
10 {
11     long thread;
12     pthread_t* thread_handles;
13
14     thread_count = strtol(argv[1], NULL, 10);
15     thread_handles = malloc(thread_count * sizeof(pthread_t));
16
17     for(thread = 0; thread < thread_count; thread++)
18         pthread_create(&thread_handles[thread], NULL, Hello, (void*)thread);
19
20     printf("Hello from the main thread\n");
21
22     for(thread = 0; thread < thread_count; thread++)
23         pthread_join(thread_handles[thread], NULL);
24
25     free(thread_handles);
26     return 0;
27 }
28
29 void *Hello(void* rank)
30 {
31     long my_rank = (long)rank;
32     printf("Hello from thread %ld of %d\n", my_rank, thread_count);
33     return NULL;
34 }
35
```

创建线程

- thread 用于存储新创建线程的唯一标识符
- attr 设置线程属性（如栈大小、调度策略等）
- start_routine 是线程的入口函数，需匹配函数签名
- arg 可传递任意类型的数据，但需强制转换

```
int pthread_create(  
    pthread_t *thread,           // 存储新线程的ID  
    const pthread_attr_t *attr, // 线程属性 (通常为NULL)  
    void *(*start_routine)(void *), // 线程入口函数  
    void *arg                    // 传递给线程函数的参数  
);
```

高级用法：线程属性定制，包括分离状态、堆栈大小、堆栈地址、调度策略、调度优先级、作用域等，可以参考：
https://www.ibm.com/docs/en/i/7.3?topic=ssw_ibm_i_73/apis/users_g4.html

终止线程（可选）

- 显式终止线程并返回状态，可在任意位置调用
- 主线程退出但保持进程存活（等待子线程）

```
void pthread_exit(  
    void *retval                // 传递线程的退出状态  
);
```

参数传递方法

- 传递：将数据转化为(void *)
- 使用：将数据转化为原有类型
- 多参数传递：定义结构体

```
void *(*start_routine)(void *)
```

```
long taskids[NUM_THREADS];

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

简单数据传递示例

```
struct thread_data{
    int  thread_id;
    int  sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
    struct thread_data *my_data;
    ...|
    my_data = (struct thread_data *) threadarg;
    ...
}

int main (int argc, char *argv[])
{
    ...
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
    ...
}
```

复杂数据传递示例

示例程序: `hello_arg2.c`

- `pthread_join` 是一种线程同步机制，它通过阻塞调用线程（通常是主线程）来等待目标线程终止，并获取其返回值

```
int pthread_join(  
    pthread_t thread, //要等待的目标线程的标识符  
    void **retval      //用于存储目标线程的返回值  
);
```

- **同步等待**：调用线程（通常是主线程）会阻塞，直到目标线程终止
- **资源回收**：确保目标线程的资源（如栈空间、线程ID）被系统回收
- **返回值传递**：获取目标线程的退出状态（通过 `retval` 参数）

```
1  #include <pthread.h>  
2  #include <stdio.h>  
3  
4  void* thread_func(void* arg) {  
5      printf("Thread running\n");  
6      return (void*)42; // 返回状态值  
7  }  
8  
9  int main() {  
10     pthread_t tid;  
11     pthread_create(&tid, NULL, thread_func, NULL);  
12  
13     void* retval;  
14     pthread_join(tid, &retval); // 主线程在此阻塞，等待子线程结束  
15     printf("Thread returned: %ld\n", (long)retval);  
16  
17     return 0;  
18 }
```

- 互斥锁 (Mutex, Mutual Exclusion) 是 Pthread 中用于保护共享资源的核心同步机制, 确保同一时间只有一个线程能访问临界区代码或数据。

- 初始化/销毁锁
- 加锁/解锁

```
// 初始化锁
int pthread_mutex_init(
    pthread_mutex_t *restrict mutex,          // 互斥锁对象
    const pthread_mutexattr_t *restrict attr  // 属性 (NULL为默认)
);

// 加锁
int pthread_mutex_lock(pthread_mutex_t *mutex);

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

```
// 动态初始化 (需手动销毁)
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);

// 静态初始化 (宏定义, 无需销毁)
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// 动态初始化的锁需手动销毁
pthread_mutex_destroy(&mutex);

pthread_mutex_lock(&mutex);    // 加锁 (阻塞直到获取锁)
// ... 临界区代码 ...
pthread_mutex_unlock(&mutex);  // 解锁
```


超算习堂 VSCode

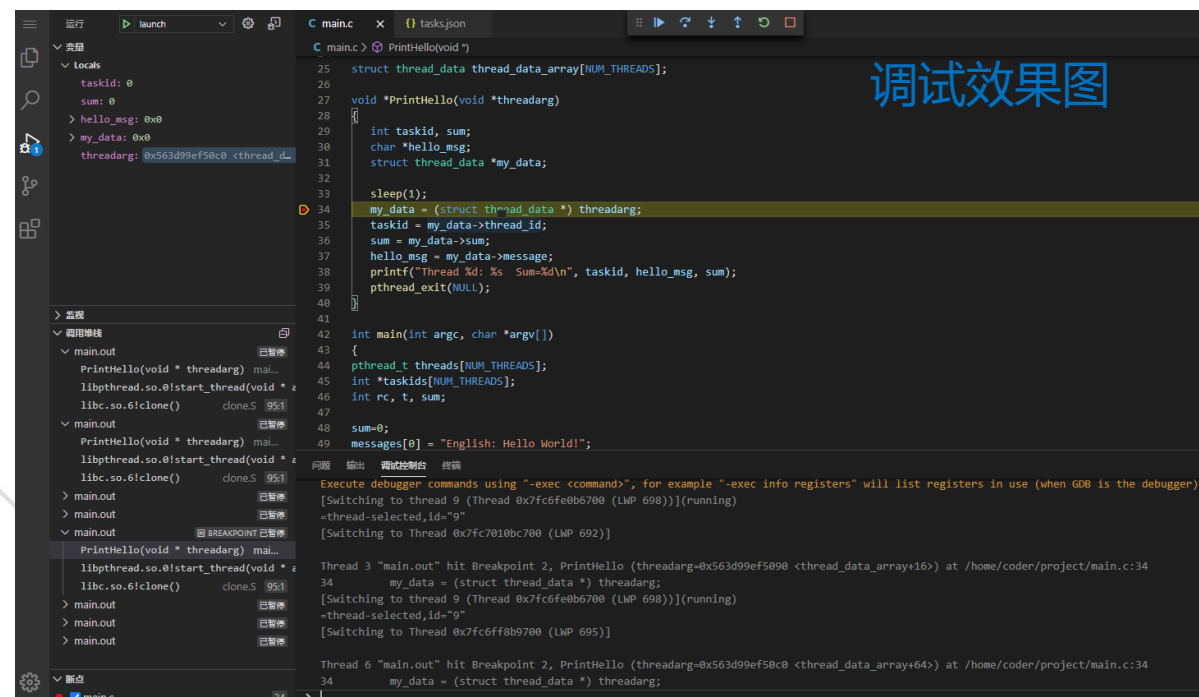
– 运行：修改settings.json

```
{} settings.json X
.vscode > {} settings.json > ...
1 {
2   "code-runner.executorMap": {
3     "c": "cd $dir && gcc $fileName -lpthread -o /tmp/cpp-compile-out/$fileNameWithoutExt.out && /tmp/cpp-compile-out/$fileNameWithoutExt.out",
4     "cpp": "cd $dir && g++ $fileName -lpthread -o /tmp/cpp-compile-out/$fileNameWithoutExt.out && /tmp/cpp-compile-out/$fileNameWithoutExt.out",
5   },
6   "extensions.autoUpdate": false
7 }
8
```

– 调试：修改tasks.json

```
{} tasks.json X
.vscode > {} tasks.json > [ ] tasks > {} 0 > [ ] args
1 {
2   "version": "2.0.0",
3   "tasks": [
4     {
5       "type": "shell",
6       "label": "build",
7       "command": "gcc",
8       "args": [
9         "-g",
10        "${file}",
11        "-o",
12        "${fileDirname}/${fileBasenameNoExtension}.out",
13        "-std=c11",
14        "-lpthread"
15      ]
16    }
17  ]
18 }
```

调试效果图



```
C main.c X {} tasks.json
25 struct thread_data thread_data_array[NUM_THREADS];
26
27 void *PrintHello(void *threadarg)
28 {
29   int taskid, sum;
30   char *hello_msg;
31   struct thread_data *my_data;
32
33   sleep(1);
34   my_data = (struct thread_data *) threadarg;
35   taskid = my_data->thread_id;
36   sum = my_data->sum;
37   hello_msg = my_data->message;
38   printf("Thread %d: %s Sum=%d\n", taskid, hello_msg, sum);
39   pthread_exit(NULL);
40 }
41
42 int main(int argc, char *argv[])
43 {
44   pthread_t threads[NUM_THREADS];
45   int *taskids[NUM_THREADS];
46   int rc, t, sum;
47
48   sum=0;
49   messages[0] = "English: Hello World!";
50
51   for(t=0; t<NUM_THREADS; t++)
52   {
53     rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
54     if (rc)
55     {
56       perror("Error: pthread_create failed");
57       exit(1);
58     }
59     taskids[t] = t;
60   }
61
62   for(t=0; t<NUM_THREADS; t++)
63   {
64     pthread_join(threads[t], NULL);
65     sum += taskids[t];
66   }
67
68   printf("Sum = %d\n", sum);
69   return 0;
70 }
```

◉ 实验1：矩阵乘法

- 使用Pthreads多线程实现并行矩阵乘法
- 设置线程数量（1-16）及矩阵规模（128-2048）
- 分析程序并行性能
- 选做：可分析不同数据及任务划分方式的影响

◉ 实验2：数组求和

- 使用Pthreads创建多线程，实现并行数组求和
- 设置线程数量（1-16）及数组规模（1M-128M）
- 分析程序并行性能及扩展性
- 选做：可分析不同聚合方式的影响

Questions?

