实验报告3

代码分析

```
void main() {
    int N=64, i, sum=0;
    #pragma omp parallel for
    for (i=0; i<N; i++)
        sum += i;
    printf("sum = %d\n", sum);
}
```

1. **并行化**：
    - 使用 `#pragma omp parallel for` 指令将 `for` 循环并行化。
    - OpenMP 会自动将循环分配给多个线程并行执行。
2. **共享变量问题**：
    - `sum` 是一个全局变量，所有线程都会访问和修改它。
    - 在并行环境中，多个线程同时对 `sum` 进行写操作会导致**数据竞争**，从而产生不确定的结果。
3. **结果不正确**：
    - 由于数据竞争，最终的 `sum` 值可能是错误的，因为多个线程对 `sum` 的更新可能会相互干扰。

```
void main() {
    int N=64, i, sum=0;
    #pragma omp parallel for reduction(+:sum)
    for (i=0; i<N; i++)
        sum += i;
    printf("sum = %d\n", sum);
}
```

1. **并行化**：
    - 同样使用 `#pragma omp parallel for` 指令将 `for` 循环并行化。
2. **使用 `reduction` 子句**：
    - `#pragma omp parallel for reduction(+:sum)` 指定了 `reduction` 子句。
    - `reduction(+:sum)` 告诉 OpenMP 编译器，`sum` 是一个归约变量，需要在多个线程之间安全地进行加法操作。
3. **线程安全的归约操作**：
    - OpenMP 会为每个线程创建一个局部的 `sum` 变量。
    - 每个线程在自己的局部变量上进行累加操作。
    - 在所有线程完成后，OpenMP 会将所有局部变量的值合并到最终的全局变量 `sum` 中。
4. **结果正确**：
    - 由于 `reduction` 子句确保了线程安全的归约操作，最终的 `sum` 值是确定且正确的。

生成访存序列

添加 `barrier` 屏障操作，当一个核心执行到 `barrier` 时停止执行，后面的操作进入核心的阻塞队列。直到所有核心都执行了 `barrier` ，即所有核心同步，所有核心从阻塞队列开始恢复执行。

- 已知存在四个核心，即：
  - 核心 0：处理 `i = 0` 到 `i = 15`
  - 核心 1：处理 `i = 16` 到 `i = 31`
  - 核心 2：处理 `i = 32` 到 `i = 47`
  - 核心 3：处理 `i = 48` 到 `i = 63`
- 对于代码一，每个核心访问相同的 `sum` 。访存队列如下：
  - `<Px, read, public_sum_addr, ->;<Px, write, public_sum_addr, public_sum_val + i>;` × 16
- 对于代码二，每个核心先访问各自的私有 `sum` 副本，然后通过 `barrier` 同步，最后归约。访存队列如下：
  - `<Px, read, Px_sum_addr, ->;<Px, write, Px_sum_addr, Px_sum_val + i>;` × 16
  - `<Px, barrier, -, ->;`
  - P0额外执行归约操作， `<P0, read, P0_sum_addr, ->;<P0, write, public_sum_addr, sum0>;<P0, read, P1_sum_addr, ->;<P0, write, public_sum_addr, sum1>;<P0, read, P2_sum_addr, ->;<P0, write, public_sum_addr, sum2>;<P0, read, P3_sum_addr, ->;<P0, write, public_sum_addr, sum3>;`

对于上述四个核心的访存队列，每个周期选取随机1-4个核心执行，即可将两段代码转换成随机的访存序列。

代码实现

添加Core类

```
class Core {
private:
    int processor_id;
    Cache *cache;
    bool barrier_flag;
    std::queue<Request> request_queue;

public:
    int *prioritiy;
    int i = getProcessorId() * 16;
    uint16_t private_sum;
    const int private_sum_addr = getProcessorId() * 0x100;
    Core(int id);
    void setCache(Cache *c) { cache = c; }
    Cache *getCache() const { return cache; }
    int getProcessorId() const { return processor_id; }
    bool getBarrierFlag() const { return barrier_flag; }
    bool isQueueEmpty() const { return request_queue.empty(); }
    int getQueueSize() const { return request_queue.size(); }
    void executeRequest(Request &request, bool omp = false, bool reduction =
false);
    void enqueueRequest(const Request &request);
    Request dequeueRequest();
    void clearBarrier() { barrier_flag = false; }
};
```

- 包含 `i`，表示循环变量
- 包含 `barrier_flag` ，表示是否执行 `barrier` 操作
- 包含 `private_sum, private_sum_addr`，表示私有sum副本的值和地址

- 包含 `request_queue`，表示核心的阻塞队列

支持barrier操作

识别和支持 `barrier` 操作：当一个核心执行到 `barrier` 时停止执行，后面的操作进入核心的阻塞队列。直到所有核心都执行了 `barrier` ，即所有核心同步，所有核心从阻塞队列开始恢复执行。

```
enum Operation {
    READ,
    WRITE,
    BARRIER         // 新增 barrier 操作
};
```

```
Operation op;
if (tokens[1] == "read") {
    op = READ;
} else if (tokens[1] == "write") {
    op = WRITE;
} else if (tokens[1] == "barrier") {
    op = BARRIER;
} else {
    std::cerr << "Invalid operation: " << tokens[1] << std::endl;
    exit(1);
}
```

```
uint16_t public_sum = 0;
void Core::executeRequest(Request &request, bool omp, bool reduction) {
    if (request.op == BARRIER) {
        barrier_flag = true;
        std::cout << "\nProcessing " << request.toString()
                  << " (Set barrier for P" << processor_id << ")" << std::endl;
        cache->print_state();
    } else {
        if (barrier_flag) {
            request_queue.push(request);
            std::cout << "\nQueued " << request.toString()
                      << " (Barrier active for P" << processor_id << ")" <<
std::endl;
        } else {
            if (request.op == READ) {
                uint16_t read_data = 0;
                cache->access(request.address, request.op, 0, &read_data);
                if (omp) {
                    private_sum = read_data;
                }
            } else {
                if (omp) {
                    if (reduction && request.address == PUBLIC_SUM_ADDR) {
                        public_sum += private_sum;
                        request.write_data = public_sum;
                    } else {
                        request.write_data = private_sum + i;
                        i++;
                    }
                }
```

```
                cache->access(request.address, request.op, request.write_data);
            }
            std::cout << "\nProcessing " << request.toString()
                      << " (Priority: " << processor_id
                      << ", Type: " << (request.op == WRITE ? "WRITE" : "READ")
<< ")" << std::endl;
            cache->print_state();
        }
    }
}
```

根据代码生成随机访存序列

```
void generateOmpRequest(const std::string& filename, bool reduction) {
    std::vector<std::queue<Request>> requests(4);
    if (reduction) {
        for (int i = 0; i < 64; i++) {
            int processor_id = i / 16;
            requests[processor_id].push(Request(processor_id, READ, processor_id
* 0x100, 0));
            requests[processor_id].push(Request(processor_id, WRITE,
processor_id * 0x100, 0));
        }
        for (int i = 0; i < 4; i++) {
            requests[i].push(Request(i, BARRIER, 0, 0));
        }
        for (int i = 0; i < 4; i++) {
            requests[0].push(Request(0, READ, i * 0x100, 0));
            requests[0].push(Request(0, WRITE, PUBLIC_SUM_ADDR, 0));
        }
    } else {
        for (int i = 0; i < 64; i++) {
            int processor_id = i / 16;
            requests[processor_id].push(Request(processor_id, READ,
PUBLIC_SUM_ADDR, 0));
            requests[processor_id].push(Request(processor_id, WRITE,
PUBLIC_SUM_ADDR, 0));
        }
    }
    std::ofstream file(filename, std::ios::out | std::ios::app);
    if (!file.is_open()) {
        std::cerr << "Failed to open or create file: " << filename << std::endl;
        return;
    }

    int ins_count = 0;
    std::vector<int> non_empty_cores = {0, 1, 2, 3};
    while (!non_empty_cores.empty()) {
        int num_active = generateRandomInt(1, non_empty_cores.size());
        ins_count += num_active;
        std::vector<int> active_cores = getRandomElements(non_empty_cores,
num_active);
        std::vector<std::string> line_requests(4, "NULL");
        for (int core : active_cores) {
            if (!requests[core].empty()) {
                line_requests[core] = requests[core].front().toString();
                requests[core].pop();
```

```
            }
        }

        for (auto it = non_empty_cores.begin(); it != non_empty_cores.end(); ) {
            if (requests[*it].empty()) {
                it = non_empty_cores.erase(it);
            } else {
                ++it;
            }
        }

        for (int i = 0; i < 4; i++) {
            file << line_requests[i];
            if (i < 3) file << ";";
            file << "\t";
        }
        file << "\n";
    }

    file.close();
    std::cout << "There are " << ins_count << " instructions" << std::endl;
    std::cout << "Generated " << filename << " successfully" << std::endl;
}
```

- 通过 `reduction` 标志，标识生成代码一或代码二的访存序列。
- 每周期从四个核心的访存序列中随机抽取1-4个队列执行。
- 因为代码一的 `write` 请求的写入值时不确定的，因此不具体设置 `write` 请求的写入值。等到程序运行时，动态的计算写入值。

总线仲裁逻辑修改

```
void Bus::arbitrate(const std::vector<Request> &requests, bool omp, bool
reduction) {
    std::vector<Request> requests_tmp = requests;
    std::vector<Request> sorted_requests;
    // 将请求加入对应处理器的请求队列
    for (Request &request : requests_tmp) {
        Core *core = cores[request.processor_id];
        core->enqueueRequest(request);
        // 如果当前处理器已设置了barrier, 则打印
        if (core->getBarrierFlag() || core->getQueueSize() > 1) {
            std::cout << "\nEnqueued " << request.toString()
                      << " (Priority: " << priorities[request.processor_id]
                      << ", Type: " << (request.op == BARRIER ? "BARRIER" :
                                        request.op == WRITE ? "WRITE" : "READ")
<< ")" << std::endl;
        }
    }
    // 遍历所有处理器，获取未设置barrier且队列不为空的请求
    for (Core* core : cores) {
        if (!core->getBarrierFlag() && !core->isQueueEmpty()) {
            sorted_requests.push_back(core->dequeueRequest());
        }
    }
    // 按 barrier > write > read 和处理器优先级排序新请求
    std::sort(sorted_requests.begin(), sorted_requests.end(),
```

```
                  [this](const Request &a, const Request &b) {
                      if (a.op != b.op) {
                          if (a.op == BARRIER) return true;
                          if (b.op == BARRIER) return false;
                          return a.op == WRITE;
                      }
                      return priorities[a.processor_id] <
priorities[b.processor_id];
                  });

    // 遍历排序后的请求
    for (Request &request : sorted_requests) {
        Core *core = cores[request.processor_id];
        core->executeRequest(request, omp, reduction);
    }

    // 检查所有核心的 barrier 状态
    bool all_barriers = allBarriersSet();
    if (all_barriers) {
        std::cout << "\nAll barriers set, clearing barriers\n";
        for (Core* core : cores) {
            core->clearBarrier();
        }
    }
}
```

- 优先级修改：屏障（`Operation::BARRIER`）> 写请求（`Operation::WRITE`）> 读请求（`Operation::READ`）。
- 核心执行了 `barrier` 操作后，后续请求进入阻塞等待队列。
- 当所有核心同步后，优先执行阻塞队列中的请求。

main函数

```
int main(int argc, char* argv[]) {
    if (argc < 2 || argc > 4) {
        std::cerr << "Usage: ./sim [-omp] [-r] filename" << std::endl;
        return 1;
    }

    bool omp_flag = false;
    bool reduction_flag = false;
    std::string filename;
    for (int i = 1; i < argc; ++i) {
        std::string arg = argv[i];
        if (arg == "-omp") {
            omp_flag = true;
        } else if (arg == "-r") {
            reduction_flag = true;
        } else {
            filename = arg;
        }
    }

    if (filename.empty()) {
        std::cerr << "Error: No filename provided." << std::endl;
        return 1;
```

```
    }

    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cout << "File does not exist. Creating and writing to " << filename
<< std::endl;
        generateOmpRequest(filename, reduction_flag);
        file.open(filename);
    }

    std::vector<Core *> cores;
    for (int i = 0; i < 4; i++) {
        Core *core = new Core(i);
        Cache *cache = new Cache(i);
        core->setCache(cache);
        cores.push_back(core);
    }

    std::vector<int> initial_priorities = {0, 1, 2, 3};
    Memory memory;
    Bus bus(cores, &memory, initial_priorities);
    for (Core* core : cores) {
        core->getCache()->setBus(&bus);
        core->getCache()->setMemory(&memory);
    }

    std::string line;
    int cycle = 0;
    while (std::getline(file, line)) {
        std::istringstream iss(line);
        std::string request_str;
        std::vector<Request> requests;

        while (std::getline(iss, request_str, ';')) {
            request_str.erase(0, request_str.find_first_not_of(" \t"));
            request_str.erase(request_str.find_last_not_of(" \t") + 1);
            if (request_str != "NULL") {
                Request request = parseRequest(request_str);
                requests.push_back(request);
            }
        }
        std::cout << "\n----------Cycle " << cycle++ << "----------\n" <<
std::endl;
        if (!requests.empty()) {
            bus.arbitrate(requests, omp_flag, reduction_flag);
        }
    }

    bool queue_empty = false;
    while (!queue_empty) {
        queue_empty = true;
        for (Core *core : cores) {
            if (!core->isQueueEmpty()) {
                queue_empty = false;
                break;
            }
        }
        if (!queue_empty) {
```

```
            std::cout << "\n----------Cycle " << cycle++ << "----------\n" << std::endl;
            bus.arbitrate(std::vector<Request>(), omp_flag, reduction_flag);
        }
    }

    file.close();
    for (Core* core : cores) {
        delete core->getCache();
        delete core;
    }
    return 0;
}
```

- 支持以下三种指令
  - `./cache_sim filename`：正常执行，`omp == false && reduction == false`
  - `./cache_sim -omp filename`：执行代码一，`omp == true && reduction == false`
  - `./cache_sim -omp -r filename`：执行代码二，`omp == true && reduction == true`
- `omp == true` 时，忽略文件中 `write` 请求的值，在执行时动态计算

测试结果

代码一

1. 执行两次 `.\cache_sim.exe -omp  ..\data\without_reduction.txt`
2. 访存序列生成（仅展示第一次生成）

```
<P0, read, 1024, ->;    <P1, read, 1024, ->;    <P2, read, 1024, ->;    <P3, read, 1024, ->
<P0, write, 1024, 0>;    <P1, write, 1024, 0>;    <P2, write, 1024, 0>;    <P3, write, 1024, 0>
NULL;    <P1, read, 1024, ->;    <P2, read, 1024, ->;    <P3, read, 1024, ->
<P0, read, 1024, ->;    <P1, write, 1024, 0>;    NULL;    <P3, write, 1024, 0>
<P0, write, 1024, 0>;    NULL;    NULL;    <P3, read, 1024, ->
<P0, read, 1024, ->;    <P1, read, 1024, ->;    <P2, write, 1024, 0>;    <P3, write, 1024, 0>
NULL;    NULL;    <P2, read, 1024, ->;    NULL
NULL;    <P1, write, 1024, 0>;    NULL;    <P3, read, 1024, ->
NULL;    <P1, read, 1024, ->;    <P2, write, 1024, 0>;    <P3, write, 1024, 0>
<P0, write, 1024, 0>;    <P1, write, 1024, 0>;    <P2, read, 1024, ->;    <P3, read, 1024, ->
NULL;    NULL;    NULL;    <P3, write, 1024, 0>
NULL;    <P1, read, 1024, ->;    NULL;    <P3, read, 1024, ->
<P0, read, 1024, ->;    <P1, write, 1024, 0>;    <P2, write, 1024, 0>;    <P3, write, 1024, 0>
<P0, write, 1024, 0>;    <P1, read, 1024, ->;    <P2, read, 1024, ->;    <P3, read, 1024, ->
<P0, read, 1024, ->;    NULL;    NULL;    NULL
<P0, write, 1024, 0>;    <P1, write, 1024, 0>;    NULL;    <P3, write, 1024, 0>
<P0, read, 1024, ->;    <P1, read, 1024, ->;    <P2, write, 1024, 0>;    NULL
<P0, write, 1024, 0>;    <P1, write, 1024, 0>;    NULL;    <P3, read, 1024, ->
<P0, read, 1024, ->;    <P1, read, 1024, ->;    <P2, read, 1024, ->;    <P3, write, 1024, 0>
NULL;    <P1, write, 1024, 0>;    NULL;    NULL
```

```
<P0, write, 1024, 0>;   <P1, read, 1024, ->;    <P2, write, 1024, 0>;   <P3, read, 1024, ->
NULL;   <P1, write, 1024, 0>;   NULL;   NULL
NULL;   <P1, read, 1024, ->;    <P2, read, 1024, ->;    NULL
NULL;   <P1, write, 1024, 0>;   <P2, write, 1024, 0>;   NULL
NULL;   NULL;   NULL;   <P3, write, 1024, 0>
NULL;   NULL;   <P2, read, 1024, ->;    NULL
NULL;   NULL;   <P2, write, 1024, 0>;   <P3, read, 1024, ->
<P0, read, 1024, ->;    <P1, read, 1024, ->;    <P2, read, 1024, ->;    <P3, write, 1024, 0>
<P0, write, 1024, 0>;   <P1, write, 1024, 0>;   <P2, write, 1024, 0>;   <P3, read, 1024, ->
<P0, read, 1024, ->;    <P1, read, 1024, ->;    NULL;   <P3, write, 1024, 0>
NULL;   <P1, write, 1024, 0>;   NULL;   NULL
<P0, write, 1024, 0>;   <P1, read, 1024, ->;    NULL;   <P3, read, 1024, ->
<P0, read, 1024, ->;    <P1, write, 1024, 0>;   <P2, read, 1024, ->;    <P3, write, 1024, 0>
<P0, write, 1024, 0>;   <P1, read, 1024, ->;    <P2, write, 1024, 0>;   NULL
NULL;   NULL;   NULL;   <P3, read, 1024, ->
<P0, read, 1024, ->;    NULL;   <P2, read, 1024, ->;    <P3, write, 1024, 0>
<P0, write, 1024, 0>;   <P1, write, 1024, 0>;   <P2, write, 1024, 0>;   <P3, read, 1024, ->
NULL;   NULL;   NULL;   <P3, write, 1024, 0>
NULL;   <P1, read, 1024, ->;    <P2, read, 1024, ->;    <P3, read, 1024, ->
<P0, read, 1024, ->;    <P1, write, 1024, 0>;   <P2, write, 1024, 0>;   <P3, write, 1024, 0>
NULL;   <P1, read, 1024, ->;    <P2, read, 1024, ->;    <P3, read, 1024, ->
<P0, write, 1024, 0>;   <P1, write, 1024, 0>;   <P2, write, 1024, 0>;   <P3, write, 1024, 0>
NULL;   NULL;   <P2, read, 1024, ->;    NULL
<P0, read, 1024, ->;    NULL;   <P2, write, 1024, 0>;   NULL
NULL;   NULL;   <P2, read, 1024, ->;    NULL
NULL;   NULL;   <P2, write, 1024, 0>;   NULL
<P0, write, 1024, 0>;   NULL;   <P2, read, 1024, ->;    NULL
NULL;   NULL;   <P2, write, 1024, 0>;   NULL
<P0, read, 1024, ->;    NULL;   NULL;   NULL
<P0, write, 1024, 0>;   NULL;   NULL;   NULL
<P0, read, 1024, ->;    NULL;   NULL;   NULL
<P0, write, 1024, 0>;   NULL;   NULL;   NULL
<P0, read, 1024, ->;    NULL;   NULL;   NULL
<P0, write, 1024, 0>;   NULL;   NULL;   NULL
```

3. 测试结果

```
#第一次结果
----------Cycle 53----------
Processing <P0, write, 1024, 1286> (Priority: 0, Type: WRITE)
Cache State (Processor 0):
Set 0:  [T:0x20 S:M D:1286 L:32]        [INVALID]
Set 1:  [INVALID]                       [INVALID]
Set 2:  [INVALID]                       [INVALID]
Set 3:  [INVALID]                       [INVALID]
Set 4:  [INVALID]                       [INVALID]
Set 5:  [INVALID]                       [INVALID]
Set 6:  [INVALID]                       [INVALID]
Set 7:  [INVALID]                       [INVALID]
```

```
#第二次结果
----------Cycle 56----------
Processing <P1, write, 1024, 1068> (Priority: 1, Type: WRITE)
Cache State (Processor 1):
Set 0:  [T:0x20 S:M D:1068 L:32]        [INVALID]
Set 1:  [INVALID]                       [INVALID]
Set 2:  [INVALID]                       [INVALID]
Set 3:  [INVALID]                       [INVALID]
Set 4:  [INVALID]                       [INVALID]
Set 5:  [INVALID]                       [INVALID]
Set 6:  [INVALID]                       [INVALID]
Set 7:  [INVALID]                       [INVALID]
```

正确结果为2016，而代码一执行结果为1286和1068，可见多个线程同时对 `sum` 进行写操作会导致冲突，从而产生不确定的结果。

代码二

1.执行两次 `.\cache_sim.exe -omp -r  ..\data\without_reduction.txt`

2.访存序列生成（仅展示第一次生成）

```
<P0, read, 0, ->;   <P1, read, 256, ->; <P2, read, 512, ->; <P3, read, 768, ->
NULL;   <P1, write, 256, 0>;    <P2, write, 512, 0>;    NULL
NULL;   NULL;   <P2, read, 512, ->; <P3, write, 768, 0>
<P0, write, 0, 0>;  NULL;   <P2, write, 512, 0>;    <P3, read, 768, ->
<P0, read, 0, ->;   NULL;   <P2, read, 512, ->; <P3, write, 768, 0>
<P0, write, 0, 0>;  <P1, read, 256, ->; <P2, write, 512, 0>;    NULL
NULL;   <P1, write, 256, 0>;    <P2, read, 512, ->; <P3, read, 768, ->
NULL;   NULL;   <P2, write, 512, 0>;    NULL
NULL;   NULL;   <P2, read, 512, ->; <P3, write, 768, 0>
<P0, read, 0, ->;   <P1, read, 256, ->; <P2, write, 512, 0>;    <P3, read, 768,
->
NULL;   <P1, write, 256, 0>;    NULL;   NULL
<P0, write, 0, 0>;  <P1, read, 256, ->; <P2, read, 512, ->; <P3, write, 768, 0>
<P0, read, 0, ->;   NULL;   NULL;   <P3, read, 768, ->
<P0, write, 0, 0>;  NULL;   <P2, write, 512, 0>;    NULL
<P0, read, 0, ->;   <P1, write, 256, 0>;    <P2, read, 512, ->; NULL
<P0, write, 0, 0>;  <P1, read, 256, ->; <P2, write, 512, 0>;    <P3, write, 768,
0>
<P0, read, 0, ->;   NULL;   NULL;   NULL
NULL;   NULL;   NULL;   <P3, read, 768, ->
<P0, write, 0, 0>;  <P1, write, 256, 0>;    <P2, read, 512, ->; <P3, write, 768,
0>
<P0, read, 0, ->;   NULL;   NULL;   NULL
NULL;   NULL;   <P2, write, 512, 0>;    NULL
NULL;   NULL;   <P2, read, 512, ->; <P3, read, 768, ->
<P0, write, 0, 0>;  <P1, read, 256, ->; <P2, write, 512, 0>;    NULL
<P0, read, 0, ->;   NULL;   NULL;   NULL
NULL;   <P1, write, 256, 0>;    <P2, read, 512, ->; <P3, write, 768, 0>
<P0, write, 0, 0>;  <P1, read, 256, ->; <P2, write, 512, 0>;    <P3, read, 768,
->
<P0, read, 0, ->;   <P1, write, 256, 0>;    NULL;   NULL
NULL;   <P1, read, 256, ->; NULL;   <P3, write, 768, 0>
NULL;   NULL;   NULL;   <P3, read, 768, ->
NULL;   <P1, write, 256, 0>;    <P2, read, 512, ->; <P3, write, 768, 0>
NULL;   <P1, read, 256, ->; <P2, write, 512, 0>;    <P3, read, 768, ->
```

```
NULL;   <P1, write, 256, 0>;     NULL;   NULL
NULL;   <P1, read, 256, ->; <P2, read, 512, ->; <P3, write, 768, 0>
<P0, write, 0, 0>;  <P1, write, 256, 0>;    <P2, write, 512, 0>;    <P3, read, 768, ->
<P0, read, 0, ->;   <P1, read, 256, ->; <P2, read, 512, ->; <P3, write, 768, 0>
NULL;   NULL;   <P2, write, 512, 0>;    <P3, read, 768, ->
<P0, write, 0, 0>;  <P1, write, 256, 0>;    <P2, read, 512, ->; NULL
<P0, read, 0, ->;   <P1, read, 256, ->; <P2, write, 512, 0>;    NULL
<P0, write, 0, 0>;  <P1, write, 256, 0>;    NULL;   NULL
<P0, read, 0, ->;   <P1, read, 256, ->; NULL;   <P3, write, 768, 0>
<P0, write, 0, 0>;  <P1, write, 256, 0>;    <P2, read, 512, ->; <P3, read, 768, ->
NULL;   <P1, read, 256, ->; <P2, write, 512, 0>;    <P3, write, 768, 0>
<P0, read, 0, ->;   NULL;   NULL;   NULL
<P0, write, 0, 0>;  <P1, write, 256, 0>;    <P2, read, 512, ->; <P3, read, 768, ->
NULL;   NULL;   NULL;   <P3, write, 768, 0>
<P0, read, 0, ->;   <P1, read, 256, ->; NULL;   NULL
<P0, write, 0, 0>;  <P1, write, 256, 0>;    <P2, write, 512, 0>;    <P3, read, 768, ->
NULL;   NULL;   <P2, barrier, -, ->;    NULL
<P0, read, 0, ->;   <P1, read, 256, ->; NULL;   <P3, write, 768, 0>
<P0, write, 0, 0>;  <P1, write, 256, 0>;    NULL;   <P3, read, 768, ->
<P0, read, 0, ->;   NULL;   NULL;   <P3, write, 768, 0>
<P0, write, 0, 0>;  NULL;   NULL;   NULL
NULL;   NULL;   NULL;   <P3, barrier, -, ->
<P0, barrier, -, ->;    <P1, barrier, -, ->;    NULL;   NULL
<P0, read, 0, ->;   NULL;   NULL;   NULL
<P0, write, 1024, 0>;   NULL;   NULL;   NULL
<P0, read, 256, ->; NULL;   NULL;   NULL
<P0, write, 1024, 0>;   NULL;   NULL;   NULL
<P0, read, 512, ->; NULL;   NULL;   NULL
<P0, write, 1024, 0>;   NULL;   NULL;   NULL
<P0, read, 768, ->; NULL;   NULL;   NULL
<P0, write, 1024, 0>;   NULL;   NULL;   NULL
```

3.测试结果

```
#第一次结果
----------Cycle 61----------
Processing <P0, write, 1024, 2016> (Priority: 0, Type: WRITE)
Cache State (Processor 0):
Set 0:  [T:0x18 S:S D:888 L:39] [T:0x20 S:M D:2016 L:40]
Set 1:  [INVALID]               [INVALID]
Set 2:  [INVALID]               [INVALID]
Set 3:  [INVALID]               [INVALID]
Set 4:  [INVALID]               [INVALID]
Set 5:  [INVALID]               [INVALID]
Set 6:  [INVALID]               [INVALID]
Set 7:  [INVALID]               [INVALID]

#第二次结果
----------Cycle 63----------
Processing <P0, write, 1024, 2016> (Priority: 0, Type: WRITE)
Cache State (Processor 0):
Set 0:  [T:0x18 S:S D:888 L:39] [T:0x20 S:M D:2016 L:40]
```

```
Set 1:  [INVALID]               [INVALID]
Set 2:  [INVALID]               [INVALID]
Set 3:  [INVALID]               [INVALID]
Set 4:  [INVALID]               [INVALID]
Set 5:  [INVALID]               [INVALID]
Set 6:  [INVALID]               [INVALID]
Set 7:  [INVALID]               [INVALID]
```

正确结果为2016，代码二执行结果两次都为2016，reduction通过线程的私有 `sum` 副本和 `barrier` 同步操作，确保了线程安全的归约，最终的 `sum` 值是确定且正确的。