

华中科技大学

2024

并行处理与体系结构 课程报告

专 业:	计算机技术
班 级:	计算机硕 2411 班
学 号:	M202474309
姓 名:	刘赟贲
电 话:	18397776726
邮 件:	liuyunben@foxmail.com
完成日期:	2025-1-14



目 录

1	内存带宽测试	1
1.1	代码编译和运行说明	1
1.2	单线程内存带宽测试	1
1.3	多线程内存带宽测试	3
2	并行编程	5
2.1	代码编译和运行说明	5
2.2	并行寻找素数	6
2.2.1	采用 SPRP 的 Miller-Rabin 素性测试	6
2.3	哲学家吃饭问题	7
2.4	并发队列的设计与实现	10
2.4.1	使用互斥锁的并发队列	10
2.4.2	使用原子操作的无锁并发队列	11
2.4.3	并发队列的测试	13
3	总结与课程建议	16
3.1	总结	16
3.2	建议	16
	参考文献	17

1 内存带宽测试

1.1 代码编译和运行说明

同课程初始代码框架有所不同，在本报告提交的源码中，项目的构建和管理方式使用了 CMake 而非 Makefile。使用如图 1.1 的方式完成构建和测试。

```
coder@f094ace882a5:~/project/membw$ cmake -S . -B build/
-- The ASM compiler identification is GNU
-- Found assembler: /usr/bin/cc
-- The C compiler identification is GNU 10.2.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/coder/project/membw/build
coder@f094ace882a5:~/project/membw$ cmake --build build
Scanning dependencies of target pmbw
[ 14%] Building C object CMakeFiles/pmbw.dir/pmbw.c.o
[ 28%] Building C object CMakeFiles/pmbw.dir/util.c.o
[ 42%] Linking C executable pmbw
[ 42%] Built target pmbw
Scanning dependencies of target membw
[ 57%] Building ASM object CMakeFiles/membw.dir/x86-asm.S.o
[ 71%] Building C object CMakeFiles/membw.dir/main.c.o
[ 85%] Building C object CMakeFiles/membw.dir/util.c.o
[100%] Linking C executable membw
[100%] Built target membw
coder@f094ace882a5:~/project/membw$
```

图 1.1 项目的构建

1.2 单线程内存带宽测试

在课程中，分别使用 C 语言和汇编语言实现了几种不同的内存拷贝函数：正向拷贝、反向拷贝和使用软件方式预读（Prefetch）的拷贝。为了便于硬件读取数据，在申请内存时考虑了对齐方式的分配器。如下实现方式，其关键在于将地址的低位向上取整，从而满足同一定数值（一般为 Cache 行大小）的对齐。

```
static char *align_up(char *ptr, int align)
{
    return (char *)(((uintptr_t)ptr + align - 1) & ~(uintptr_t)(align - 1));
}
```

正向拷贝时，由于顺序地加载需要读写的数据，硬件可以很容易地从 Cache 中取出相应内容，因此效率上相对占优。反向拷贝时，由于读写顺序同硬件期待

的顺序不一致，Cache 命中率下降，故而在效率上不占优势。而使用预读的读取方式时，由于预先加载了锁需要的内容，Cache 访问命中率最高，因此是三种方式最优的一种。使用汇编语言时，使用 Intel 的 SIMD 指令集 AVX，在功能和实现上均同使用纯 C 语言的方式一致。但是，单次拷贝的位宽从 64 位变成了 128 位，因此效率明显高出前者使用纯 C 语言的方式。

测试结果如图 1.2 所示。请注意，课程提供的初始框架添加了 O2 级别的优化，这在很大程度上干扰了实验的结果。其原因在于，使用 O2 编译后，编译器完成了对于函数逻辑的优化，如图 1.3 所示，在 Godbolt 平台的使用同样编译器并开启 O2 优化的输出，在指令中拷贝已经转换成了调用 `memcpy`。

```

coder@f094ace882a5:~/project/membw$ ./build/membw
./build/membw: benchmark for memory bandwidth

=====
== Memory bandwidth tests ==
== Note: If sample standard deviation exceeds 0.1%, it's shown in brackets. ==
=====

C copy                : 3269.5 MB/s (2.1%)
C copy backwards      : 2857.2 MB/s
C copy prefetched     : 5356.5 MB/s (0.6%)
C fill                : 3701.2 MB/s (0.8%)
C memcpy()            : 17220.1 MB/s (0.8%)
SSE2 copy             : 12531.9 MB/s (1.2%)
SSE2 nontemporal copy : 17278.3 MB/s (0.7%)
SSE2 nontemporal copy prefetched : 17307.0 MB/s (0.6%)
SSE2 nontemporal fill : 17327.3 MB/s (0.7%)
coder@f094ace882a5:~/project/membw$

```

(a) 无 O2 优化

```

coder@f094ace882a5:~/project/membw$ ./build/membw
./build/membw: benchmark for memory bandwidth

=====
== Memory bandwidth tests ==
== Note: If sample standard deviation exceeds 0.1%, it's shown in brackets. ==
=====

C copy                : 17265.9 MB/s (2.3%)
C copy backwards      : 17278.9 MB/s (0.6%)
C copy prefetched     : 11910.7 MB/s (0.7%)
C fill                : 12494.2 MB/s (0.5%)
C memcpy()            : 17301.2 MB/s (1.0%)
SSE2 copy             : 12686.7 MB/s (1.2%)
SSE2 nontemporal copy : 17250.3 MB/s (0.7%)
SSE2 nontemporal copy prefetched : 17329.3 MB/s (0.8%)
SSE2 nontemporal fill : 17321.2 MB/s (1.7%)
coder@f094ace882a5:~/project/membw$ ./build/pmbw 4
pmbw (4 threads) : 42021.2 MB/s (0.7%)
coder@f094ace882a5:~/project/membw$

```

(b) 使用了 O2 优化

图 1.2 单线程内存带宽测试结果 (a)无 O2 优化和(b)使用了 O2 优化

```

void aligned_block_copy(int64_t * __restrict dst,
                       int64_t * __restrict src,
                       int size)
{
    assert(size % sizeof(int64_t) == 0);
    size /= sizeof(int64_t);
    for (int i = 0; i < size; i++) dst[i] = src[i];
}

void aligned_block_copy_backwards(int64_t * __restrict dst,
                                  int64_t * __restrict src,
                                  int size)
{
    assert(size % sizeof(int64_t) == 0);
    size /= sizeof(int64_t);
    for (int i = size - 1; i >= 0; i--) dst[i] = src[i];
}

```

```

7 aligned_block_copy(long*, long*, int):
8     test    dl, 7
9     jne     .L8
10    movsx   rax, edx
11    shr     rax, 3
12    test    eax, eax
13    jle     .L1
14    sub     eax, 1
15    lea     rdx, [8+rax*8]
16    jmp     memcpy

```

图 1.3 在 Godbolt 中同版本 GCC 其开启 O2 优化的汇编结果

1.3 多线程内存带宽测试

单个 CPU 核心无法占满 CPU 的内存带宽，因此需要使用并行的方式让多个核心同时进行带宽测试，以得到设备的内存带宽。在并行测试内存带宽时，只需要让多个线程均执行同串行测试一致的程序，然后将循环的轮次作为返回值移交给统计的线程。

在不考虑架构本身的前提下，使用 POSIX 库提供的接口实现并行，此处使用了同串行一致的指标，即带宽和标准差。测试结果如图 1.4(a)所示，系统在 4 线程左右达到带宽 42GB/s。但同时，也可以注意到测试结果的标准差较大，这反映了测试结果的不稳定。一定程度上，这是由于线程的调度造成了线程切换 CPU 执行，除了调度本身的开销，这可能带来额外的访存延迟。

```

coder@f094ace882a5:~/project/membw$ ./build/pmbw 4
pmbw (4 threads) : 42053.7 MB/s (23.5%)
coder@f094ace882a5:~/project/membw$ ./build/pmbw 4
pmbw (4 threads) : 42297.2 MB/s (21.7%)
coder@f094ace882a5:~/project/membw$ ./build/pmbw 3
pmbw (3 threads) : 38108.0 MB/s (20.7%)
coder@f094ace882a5:~/project/membw$ ./build/pmbw 2
pmbw (2 threads) : 29693.7 MB/s (9.3%)

```

(a)

```

coder@f094ace882a5:~/project/membw$ ./build/pmbw 4
pmbw (4 threads) : 41346.8 MB/s (0.3%)
coder@f094ace882a5:~/project/membw$ ./build/pmbw 4
pmbw (4 threads) : 41968.3 MB/s (1.0%)
coder@f094ace882a5:~/project/membw$ ./build/pmbw 5
pmbw (5 threads) : 42194.4 MB/s (1.2%)

```

(b)

图 1.4 绑定 CPU 前(a) 和绑定 CPU 后(b)的并行测试结果

使用 `pthread_setaffinity_np` 可以绑定线程运行的 CPU-id，从而减少由线程切换带来的开销。在分配 CPU-id 的策略上，由于不同设备的 CPU-id 的编号规则不一致，一些型号的 CPU 采用了交错的编号方式，但也存在按照连续区间编号的

情况。使用 `lscpu` 可看到 CPU 的相关信息，对于测试环境将其结果拓扑成如图 1.5 所示结构。该双 CPU 环境一共有 56 个 CPU 核心（26 个物理核心使用超线程技术）。其中 NUMA node0 的 CPU 编号位 0-13 和 28-41，剩下的为 NUMA node2 锁持有。分配时，分配同一物理核心时效率要小于分配到不同的物理核心，4 线程测试结果如图 1.4(b)所示，更多线程的测试如图 1.6 所示。

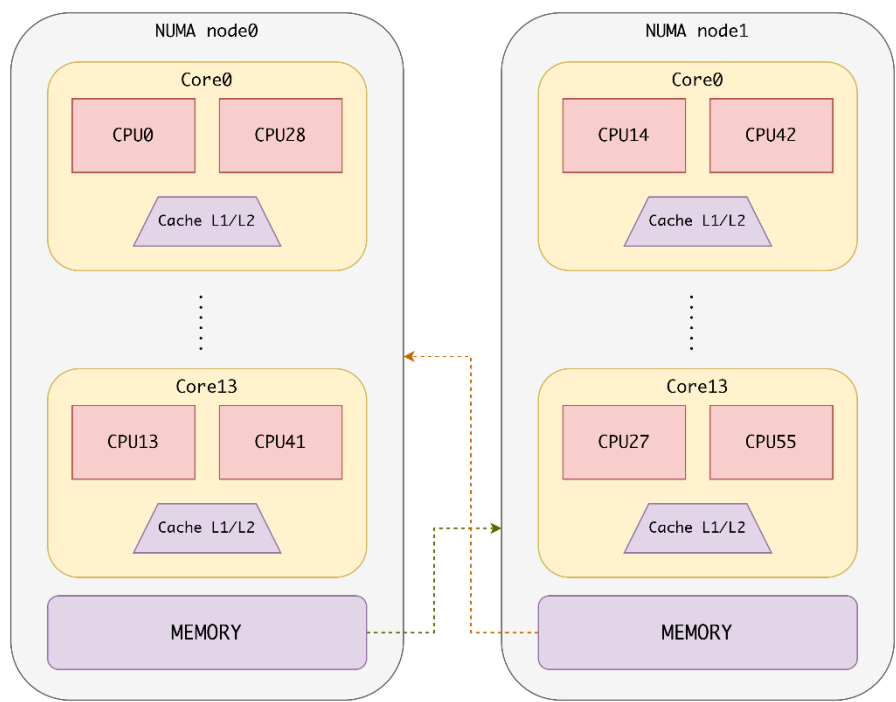


图 1.5 测试环境 CPU 拓扑结构

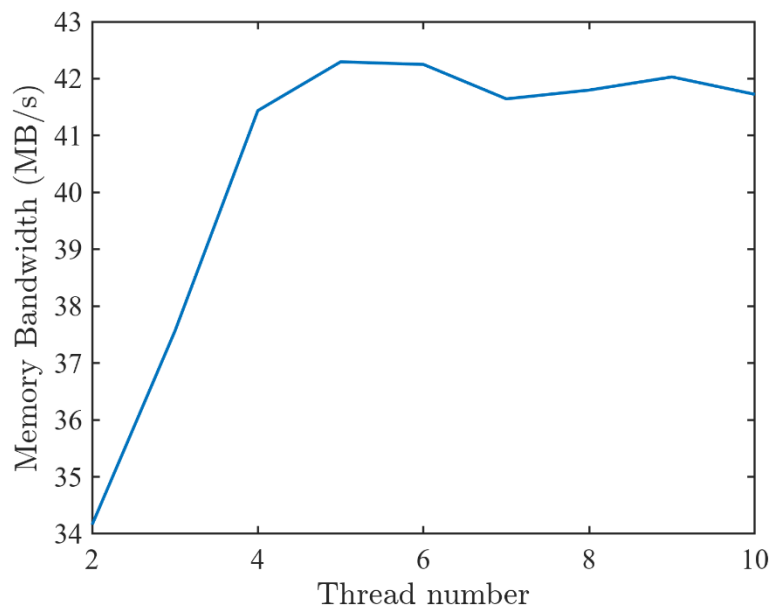


图 1.6 多线程并行内存带宽测试

2 并行编程

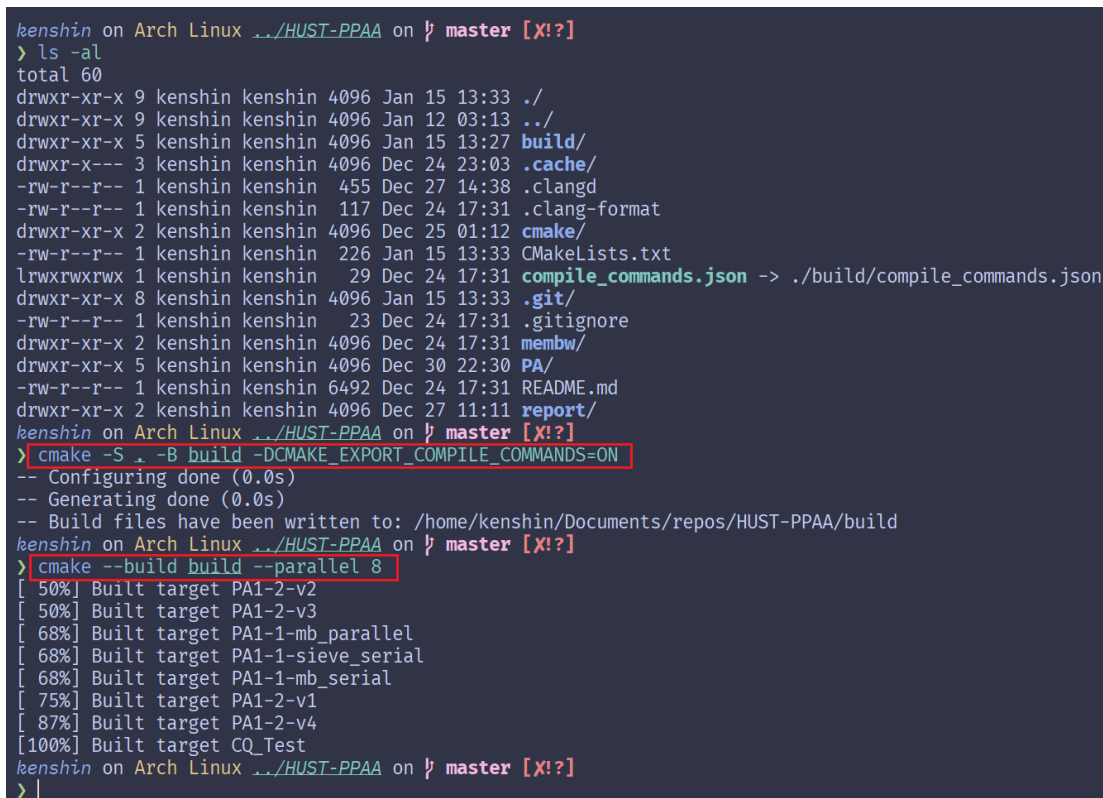
2.1 代码编译和运行说明

同第 1 节中的方式一致，本次实验采用 CMake 管理程序的构建过程。在本节中的编程语言为 C++，需要 GCC 版本大于等于 13 以满足 C++20 标准（低版本 GCC 使用 `-std=c++20` 或 `c++2a` 的方式，但需确保 `libstdc++` 版本大于 12）。对于 CMake 的版本限制至少需要 3.12 以支持 `FetchContent` 功能，用以拉取 oneTBB 源码进行构建和支持并行编译。

```
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

```
cmake --build build --parallel 8
```

使用上述两条命令在根目录下执行项目构建和编译，对应的 `target` 即为本实验中的全部测试程序，流程如图 2.1 所示。



```
kenshin on Arch Linux .../HUST-PPAA on master [X!?:]
> ls -al
total 60
drwxr-xr-x 9 kenshin kenshin 4096 Jan 15 13:33 ./
drwxr-xr-x 9 kenshin kenshin 4096 Jan 12 03:13 ../
drwxr-xr-x 5 kenshin kenshin 4096 Jan 15 13:27 build/
drwxr-xr-x 3 kenshin kenshin 4096 Dec 24 23:03 .cache/
-rw-r--r-- 1 kenshin kenshin 455 Dec 27 14:38 .clangd
-rw-r--r-- 1 kenshin kenshin 117 Dec 24 17:31 .clang-format
drwxr-xr-x 2 kenshin kenshin 4096 Dec 25 01:12 cmake/
-rw-r--r-- 1 kenshin kenshin 226 Jan 15 13:33 CMakeLists.txt
lrwxrwxrwx 1 kenshin kenshin 29 Dec 24 17:31 compile_commands.json -> ./build/compile_commands.json
drwxr-xr-x 8 kenshin kenshin 4096 Jan 15 13:33 .git/
-rw-r--r-- 1 kenshin kenshin 23 Dec 24 17:31 .gitignore
drwxr-xr-x 2 kenshin kenshin 4096 Dec 24 17:31 membw/
drwxr-xr-x 5 kenshin kenshin 4096 Dec 30 22:30 PA/
-rw-r--r-- 1 kenshin kenshin 6492 Dec 24 17:31 README.md
drwxr-xr-x 2 kenshin kenshin 4096 Dec 27 11:11 report/
kenshin on Arch Linux .../HUST-PPAA on master [X!?:]
> cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
-- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: /home/kenshin/Documents/repos/HUST-PPAA/build
kenshin on Arch Linux .../HUST-PPAA on master [X!?:]
> cmake --build build --parallel 8
[ 50%] Built target PA1-2-v2
[ 50%] Built target PA1-2-v3
[ 68%] Built target PA1-1-mb_parallel
[ 68%] Built target PA1-1-sieve_serial
[ 68%] Built target PA1-1-mb_serial
[ 75%] Built target PA1-2-v1
[ 87%] Built target PA1-2-v4
[100%] Built target CQ_Test
kenshin on Arch Linux .../HUST-PPAA on master [X!?:]
> |
```

图 2.1 课程项目的构建示例

2.2 并行寻找素数

在素数判定上,采用试除法可以在 $O(\sqrt{N})$ 的时间复杂度内判定正整数 N 是否为素数。但是这种方式下,若整数 N 存在较小的素因子则可以很快遍历到这一情况从而判定为合数;若整数 N 的最小素因子在数值上较大或是其本身就是素数,则需要较长的执行时间。综上,采用试除法的判定在数据上无关,可以拥有很高的并行程度,但任务如何均匀分配到不同线程则是重点。均衡分配任务的方法有多种,如:

(1) 使用线程池,通过任务窃取的方式达到负载均衡。在 C++11 中引入了异步执行框架(`std::future` 和 `std::async`),按照这种方式替换 `std::thread` 或 `std::jthread` 便能把负载均衡的任务交由操作系统完成。

(2) 随机化每个线程需要判定的整数集合。随机任务能够在很大程度上平均线程的计算量。但随机带来的 Shuffle 操作需要时间、也占用了一定的内存。

(3) 采用并行时更均衡的其它的素数判定方法。筛法在素数判定上有不错的执行效率,但其数据依赖很强(即必须完成所有小于 N 的数的筛操作,才能最终判定 N 的素性),采用流水线方式的并行能够提升的效果十分有限。为了获得同试除法一样的并行度,算法的选择需要满足数据无关,而 Miller-Rabin 素性判定恰好可以满足需要。但由于这是一种概率算法,执行时间并不稳定,实验中采用了一种变体,即使用强伪素数判定、以牺牲算法的一般性来确定性地判断一定区间内整数的素性。

2.2.1 采用 SPRP 的 Miller-Rabin 素性测试

Miller-Rabin 素性测试本身是一种概率素性测试,通过这种方式得出为素数的结论存在一定的错误概率,而得出为合数的结论则是完全可信的。该算法的原理基于费马小定理:如果 p 是素数,则对于任意整数 a 均有 $a^p \equiv a \pmod{p}$;若 $a \in \mathbb{Z}_p$ 即 a 在整数模 p 形成的有限域中,可得 $a^{p-1} - 1 \equiv 0 \pmod{p}$ 。

对于给定的奇数 $N = k \times 2^t + 1$,若 N 是素数,则根据费马小定理可知:

$$\forall a \in \mathbb{Z}_N, a^{N-1} - 1 = a^{k \times 2^t} - 1 = (a^k - 1) \prod_{i=0}^{t-1} (a^{k \times 2^i} + 1) \equiv 0 \pmod{N}$$

$$\Rightarrow N \mid (a^k - 1) \vee N \mid (a^{k \times 2^i} + 1), i = 0, 1, 2, \dots, t - 1$$

Miller-Rabin 测试就是基于验证这一系列条件的成立与否，类似的也有费马素性测试（检验费马小定理是否成立）。但这些并非素性的充分条件而仅仅是必要，因此很可能存在对于某一个选择的数 a 有合数通过了测试的情况。概率上可以证明，随机化的 Miller-Rabin 测试的单次失败概率小于 $1/4$ ，因此只要测试足够充分便能得到很可信的回答。

基于概率的测试并不完备，不过只要数 a 的选择得当，便能在一定范围能得到确定性的测试，研究人员将这种方法称为 SPRP。在 1993 年，Gerhard Jaeschke 发现了一组 SPRP- a 的取值集合 $\{2, 7, 61\}$ ，能够判定所有 $N \leq 4'759'123'141$ 情况的素性，在本实验中显然能够满足要求。由于判定的条件只有 $\log_2 N$ 种，素性测试的执行时间差异很小，任务的负载能够获得很好的均衡。如图 2.1 所示，多线程运行时间都较为均衡，且整体相较于串行程序加速了约 3.5 倍。

```
kenshin on Arch Linux ../HUST-PPAA on
> ./build/PA/PA1-1/PA1-1-mb_parallel
Thread #2: running time 931ms
Thread #0: running time 946ms
Thread #1: running time 950ms
Thread #4: running time 953ms
Thread #3: running time 957ms
Thread #6: running time 959ms
Thread #7: running time 963ms
Thread #5: running time 964ms
Running time = 964ms
Prime number counter = 5761455
Prime number sum = 279209790387276
Top ten maximum primes:
99999787
99999821
99999827
99999839
99999847
99999931
99999941
99999959
99999971
99999989
```

(a)

```
kenshin on Arch Linux ../HUST-PPAA
> ./build/PA/PA1-1/PA1-1-mb_serial
Running time = 3582ms
Prime number counter = 5761455
Prime number sum = 279209790387276
Top ten maximum primes:
99999787
99999821
99999827
99999839
99999847
99999931
99999941
99999959
99999971
99999989
```

(b)

图 2.1 并行(a)和串行(b)的 Miller-Rabin-SPRP 测试结果

2.3 哲学家吃饭问题

哲学家吃饭问题（Dining Philosophers Problem）是一个经典的同步问题，常

用于讨论并发编程中的资源共享和死锁问题。这个问题由 Edsger Dijkstra 在 1965 年提出，用来描述多个进程或线程如何安全地共享资源。实验中使用的编程语言为 C++，采取如下的方式进行抽象，

```
struct Chopstick : std::binary_semaphore {
    explicit Chopstick() : std::binary_semaphore{1} {}
};

std::vector<Chopstick> chopstick(N);    // 筷子 -> 二元信号量
std::vector<std::jthread> philosophers; // 哲学家 -> 线程
```

哲学家（编号为 i ）的就餐过程的抽象如下，

```
auto event = [&chopstick](const std::stop_token& token, int i) {
    int L = i;
    int R = (i + 1) % N;

    while (!token.stop_requested()) {
        chopstick[L].acquire();
        chopstick[R].acquire();
        // 就餐
        chopstick[L].release();
        chopstick[R].release();

        std::this_thread::yield();
    }
};
```

上述的版本会因为存在不当的资源获取顺序而发生死锁，如所有哲学家均已经持有左手的筷子，同时等待右侧的筷子，因此形成了死锁。解决方案有，（版本 2）限制其中一位哲学家的就餐，让剩余的人中至少有一位哲学家能够满足就餐条件；（版本 3）或是更改资源的获取顺序，如要求获取资源的编号为 0-4，哲学家必须尝试先获得编号小的筷子。如此以来，便不会发生死锁。但是显而易见的，由于 0 号资源会被激烈的竞争，在附近的哲学家的就餐条件相较于远离 0 号资源的哲学家要更为严苛，最终导致资源的分配不公。考虑到语言特性，C++ 的 `std::lock` 的设计可以将多个锁一起以安全的方式上锁，（版本 4）采用了延迟上锁机制，只有在获取全部资源后再尝试上锁。

在源码文件中，有一个使用 Python 实现的 checker.py 用以统计每个哲学家的就餐次数。测试结果如图 2.2 到 2.5 所示。

```
kenshin on Arch Linux .. /HUST-PPAA on ʘ master [X!?] took 2s
> ./PA/PA1-2/checker.py ./build/PA/PA1-2/PA1-2-v1
The implementation of ./build/PA/PA1-2/PA1-2-v1 causes DEADLOCK.
Press [ENTER] to continue.

Philosopher 0 dined      39 times.
Philosopher 1 dined      21 times.
Philosopher 2 dined       0 times.
Philosopher 3 dined      18 times.
Philosopher 4 dined       0 times.
```

图 2.2 版本 1，产生死锁的测试结果

```
kenshin on Arch Linux .. /HUST-PPAA on ʘ master [X!?] took 49s
> ./PA/PA1-2/checker.py ./build/PA/PA1-2/PA1-2-v2
The implementation of ./build/PA/PA1-2/PA1-2-v2 causes STARVATION.
Press [ENTER] to continue.

Philosopher 0 dined       0 times.
Philosopher 1 dined    14213 times.
Philosopher 2 dined    9706 times.
Philosopher 3 dined   10367 times.
Philosopher 4 dined   27649 times.
```

图 2.3 版本 2，使哲学家饥饿的测试结果

```
kenshin on Arch Linux .. /HUST-PPAA on ʘ master [X!?] took 36s
> ./PA/PA1-2/checker.py ./build/PA/PA1-2/PA1-2-v3
The implementation (Resource Hierarchy) of ./build/PA/PA1-2/PA1-2-v3 is UNFAIR.
Press [ENTER] to continue.

Philosopher 0 dined     3643 times.
Philosopher 1 dined    11553 times.
Philosopher 2 dined    30154 times.
Philosopher 3 dined    62954 times.
Philosopher 4 dined     4041 times.
```

图 2.4 版本 3，更改资源获取顺序的测试结果

```
kenshin on Arch Linux .. /HUST-PPAA on ʘ master [X!?] took 21s
> ./PA/PA1-2/checker.py ./build/PA/PA1-2/PA1-2-v4 6
Philosopher 0 dined     5445 times.
Philosopher 1 dined     5328 times.
Philosopher 2 dined     5394 times.
Philosopher 3 dined     5878 times.
Philosopher 4 dined     5376 times.
Philosopher 5 dined     5314 times.
```

图 2.5 版本 4，可变参数的延迟上锁的测试结果

2.4 并发队列的设计与实现

在后续报告中，实现了两种队列，其一是使用了互斥锁的单生产者和单消费者（SPSC）并发队列；另一则是使用原子量和内存屏障的无锁并发队列，可支持多生产者和多消费者（MPMC）场景。二者的共有的实现逻辑（即基本的队列结构）在此描述如下：

(1) 数据存储的结构均为“有界环形缓冲区”，队列循环地使用一次性申请的内存，不再扩容和动态分配数据节点。数据存储结构在实现时使用了 C++ 的 `std::array`。正因数据一次性分配，无锁并发队列在数据上需要增加了一个原子量 `version` 作为版本指示器以处理 ABA 问题。

(2) 由于使用 `size_t` 作为队首和队尾的指示器，在获取下标时需要进行取模运算，设计中强制初始获取的数组长度为 2 的幂次，并通过这一特性将取模转变成按位与操作，以减小除法运算带来的开销。同时，由于同余关系，采用这种方式也避免了 `size_t` 自然溢出的问题。

$$index \bmod 2^n - 1 = index \text{ bitAnd } 2^n - 1 \quad (2-1)$$

(3) 在接口上，为了便于同 `tbb::concurrent_queue` 测试的便利，队列的入队和出队没有遵循文档的要求，而是同一使用 `push` 作为入队、`pop` 作为出队。对比对象也变更为 `tbb::concurrent_bounded_queue`，以满足相同测试条件。

2.4.1 使用互斥锁的并发队列

在确保数据可访问性的前提下，入队和出队可以同时执行。在队首和队尾均使用一把互斥锁进行管理，确保同一时刻最多只有一个入队和一个出队操作。为了确保数据可使用性，采用信号量进行管理可入队和可出队的数据数目。

```
void push(const T& value) noexcept {
    std::lock_guard lock(mu_tail_);
    const std::size_t index = (tail_ += 1U) & MOD_MASK;
    sem_enqueue_.acquire();
    buffer_[index] = value;
    sem_dequeue_.release();
}

void pop(T& value) noexcept {
    std::lock_guard lock(mu_head_);
    const std::size_t index = (head_ += 1U) & MOD_MASK;
```

```

sem_dequeue_.acquire();
value = buffer_[index];
sem_enqueue_.release();
}

```

此处，以入队操作为例，调用函数的线程首先要申请获得对应的互斥锁，而后通过入队信号量检查是否有空的位置，并带着锁阻塞到能够入队为止。同样的流程在出队操作上也保持一致。由于自然溢出和取模 $2^n - 1$ 的效果类似，容易发现，该过程中 `head_` 或 `tail_` 的自然溢出并不会影响程序的正确性，算法总是能正确得到正确的位置。

2.4.2 使用原子操作的无锁并发队列

本设计中，无锁队列使用原子操作和内存屏障，允许线程在没有互斥锁的情况下安全地进行入队和出队操作。

实现上，通过版本号，队列能够确保在多个线程同时访问同一数据块时的数据安全，进而处理 ABA 问题。此处使用利用如图 2.6 所示的循环序列的值，来标识能够操作的线程应该持有的 `tail/head` 版本。考虑到该等差序列的公差也为 $2^n - 1$ ，因此可以安全地让其自然溢出，该序列地长度也确保了线程阻塞期间不会完成多轮读写。

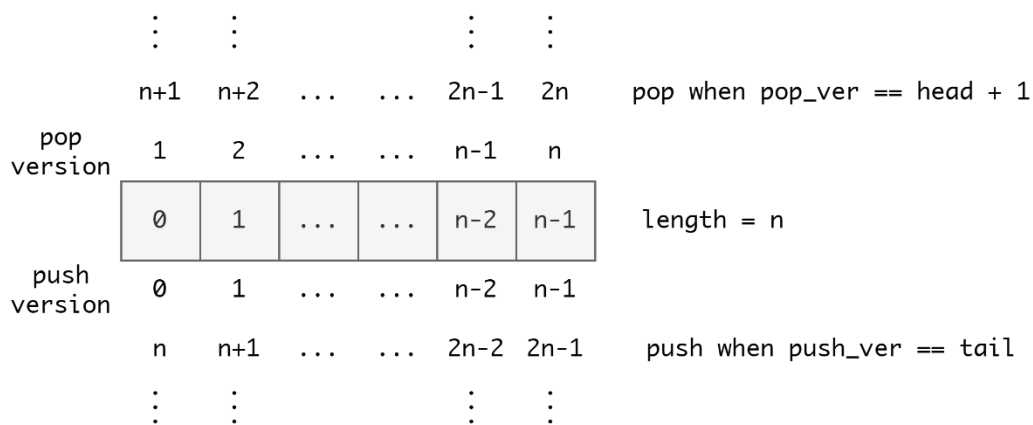


图 2.6 循环的等差序列（版本控制标识）

无锁队列的头尾指针使用两个原子变量，即 `head` 和 `tail` 分别跟踪队列的头部和尾部。队列的入队和出队的算法流程如表 2-1 和 2-2 所示。简单而言，算法在入队/出队时通过 `version` 和 `tail/head` 当前值的匹配关系（使用）来确定是否能

入队/出队，然后利用 CAS 尝试让队列的 tail/head 自增 1 的步长来使当前线程占用该位置，CAS 失败则更新线程需要抢占的下一个节点的下标。

表 2-1 无锁队列的入队算法流程

Algorithm 2-1: MPMC Queue push
Data: <i>data</i> to push repeat <i>tail</i> = Queue.tail.load() <i>index</i> = Get buffer index of <i>tail</i> <i>version</i> = buffer[<i>index</i>].version() if <i>version</i> = <i>tail</i> then <i>state</i> = Queue.tail.CompareAndExchange(<i>tail</i> , <i>tail</i> + 1) if <i>state</i> denotes Queue.tail equals to <i>tail</i> then break; end until current index of buffer is empty; buffer[<i>index</i>].value = <i>data</i> buffer[<i>index</i>].version.store(<i>tail</i> + 1)// version for pop

表 2-2 无锁队列的出队算法流程

Algorithm 2-2: MPMC Queue pop
Data: <i>data</i> reference to pop repeat <i>head</i> = Queue.head.load() <i>index</i> = Get buffer index of <i>head</i> <i>version</i> = buffer[<i>index</i>].version() if <i>version</i> = <i>head</i> + 1 then <i>state</i> = Queue.head.CompareAndExchange(<i>head</i> , <i>head</i> + 1) if <i>state</i> denotes Queue.head equals to <i>head</i> then break; end until current index of buffer has value; <i>data</i> = buffer[<i>index</i>].value buffer[<i>index</i>].version.store(<i>head</i> + buffer.length)// version for push

不难发现，在确保不会有多个入队/出队线程同时访问的情况下，仅使用一个原子布尔变量来标记也可行。不过，一旦缓冲区长度不足，则有很大概率出现 ABA 问题，即某一线程在获取位置后被阻塞，然后在无法确认的情况下覆盖了当前节点的值。

代码在具体实现时使用了 C++ 编程语言，在内存模型上还有两个细节：

(1) 对于原子量的操作并不一定保证无锁，在内存不与 Cache 行对齐的情况下，对于原子量的访问可能需要多次访存，造成不必要的性能衰减。因此，虽然会在测试时带来不公平的因素，但使用 alignas(CACHE_LINE_SIZE)是必要的。

(2) 多线程访问同一原子量时存在内存定序问题，即多线程针对同一原子量的操作的读写顺序可能由于指令的乱序执行而产生不符合预期的结果。代码实现中使用了宽松定序（即只需要保证原子性）和获取-释放定序（与内存屏障功能类似）两类。

2.4.3 并发队列的测试

```
kenshin on Arch Linux ../HUST-PPAA on  master [X!?:]
> ./build/PA/PA2/CQ_Test
constexpr auto internal::measure(int, int) [with QueueType = tbb::detail::d2::concurrent_bounded_queue<int>]
Running time: 1589ms
Operations per second: 47.51 M
constexpr auto internal::measure(int, int) [with QueueType = PA2::BLOCKING_QUEUE<int>]
Running time: 4589ms
Operations per second: 16.45 M
constexpr auto internal::measure(int, int) [with QueueType = PA2::NONBLOCKING_QUEUE<int>]
Running time: 1147ms
Operations per second: 65.82 M
kenshin on Arch Linux ../HUST-PPAA on  master [X!?:] took 7s
> █
```

图 2.7 运行测试程序的输出

如图 2.7 所示，执行测试程序后，可得到各类并发队列完成一定数量的入队和出队操作的运行时间和平均每秒能执行的操作数目。该测试程序利用随机数生成器入队的数据，然后在出队时计算元素和进行比较，在一定程度上能够验证这些并发队列的算法正确性。

在随机数生成上，本项目并未遵循课程建议的“先为每个线程生成待入队的随机数序列，然后再进行测试”的方案，而是采用了为每个入队测试的线程都生成一份独有的、线程安全的随机数生成器。C 语言中提供的 `rand()` 函数并非线程安全，且不能作为对象创建多份，但在 C++ 中随机数生成器以对象的形式存在，故而可以满足上述的需求。此处，替换 `rand()` 为如下源码的实现方案，利用 `thread_local` 关键字生成线程独有的随机数生成器，从而实现无锁并发。

```
inline auto rand_int() noexcept -> int {
    thread_local std::mt19937 engine(std::random_device{}());
    thread_local std::uniform_int_distribution<int> dist(INT_MIN, NT_MAX);
    return dist(engine);
}
```

考虑不同线程数的测试，其结果如图 2.8、图 2.9 和图 2.10 所示。在本项目实现的无锁并发队列中，由于采用了对齐的内存分配方式，因而在并发压力小的时候其效率远超有其它并发队列，这并非由算法的高效带来，而是硬件优化的结

果。不过，在随着线程数增大，可以发现该无锁队列的效率同 `oneTBB` 的实现在效率上比较接近。

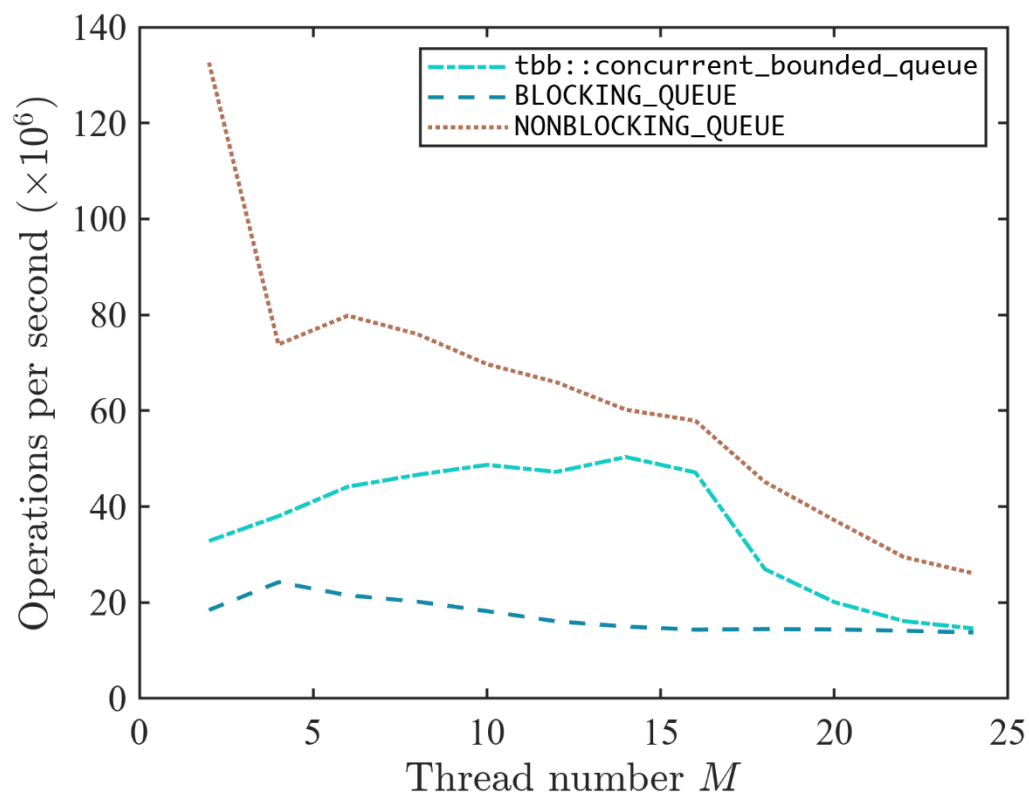


图 2.8 执行效率随线程变化情况（生产者、消费者比例 1:1）

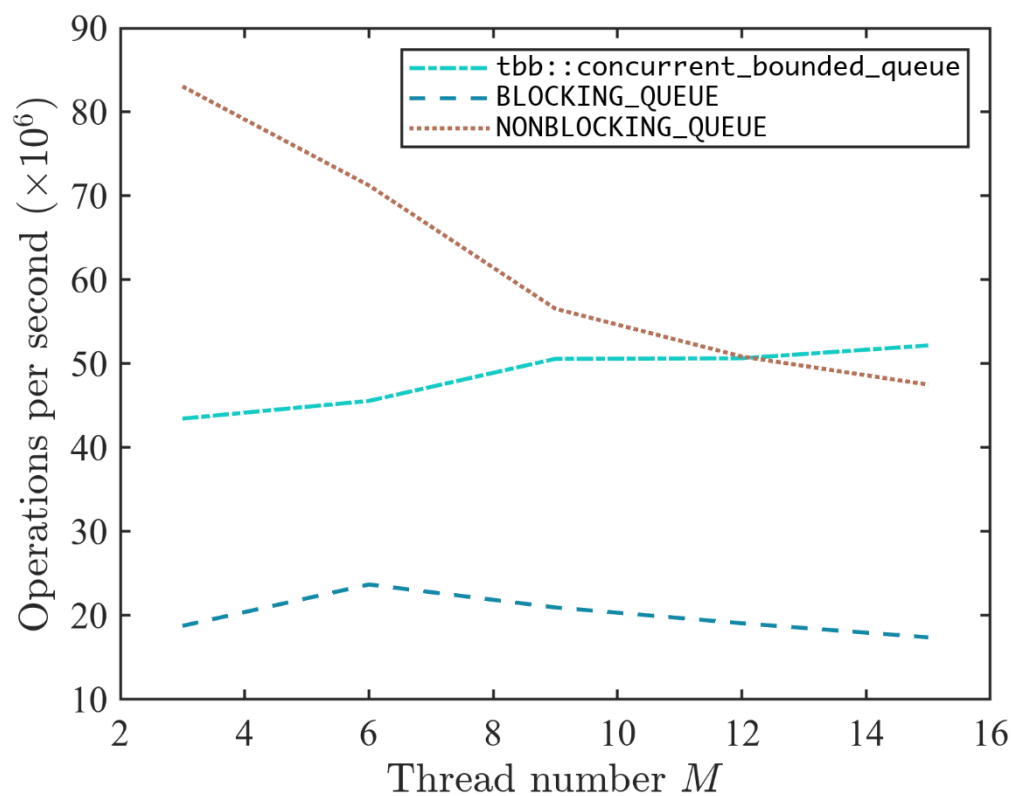


图 2.9 执行效率随线程变化情况（生产者、消费者比例 1:2）

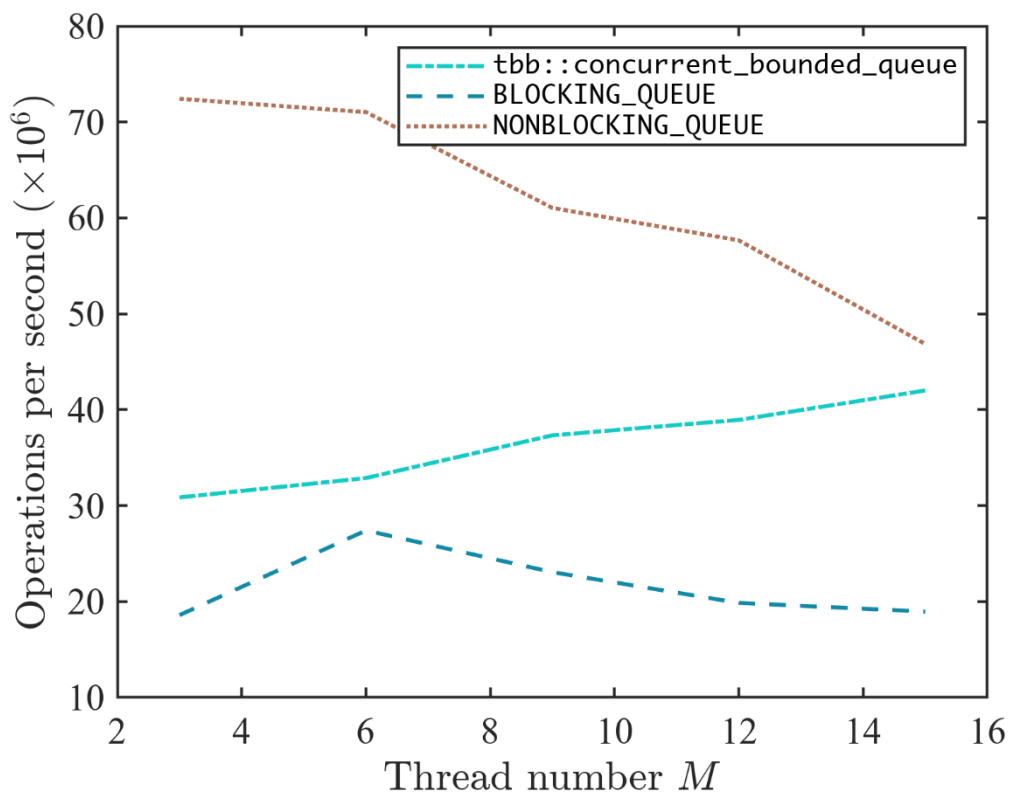


图 2.10 执行效率随线程变化情况（生产者、消费者比例 2:1）

此外，从图 2.8 中可以发现，当线程数超过了硬件能够支持的最大线程数（测试环境最大支持 16 线程），线程切换的代价增大，并发队列的性能逐步降低。

3 总结与课程建议

3.1 总结

本课程中，我尝试了使用 C++ 进行并行编程，也跟随姚老师的讲解在多处理器 NUMA 架构中优化程序，总的来说收获颇丰。

在内存带宽测试中，我尝试过计算其理论带宽，发现理论最高应当为 $2133\text{MHz} \times 8\text{B} \times 2\text{CPU} \times 2\text{channel} = 68\text{GB/s}$ ，但测试结果显得小得多。在查阅资料时发现，由于种种因素 CPU 的内存控制器难以达到这么高的带宽，而一般认为这里实际的带宽为理论的 80%（约为 54.4GB/s）。但测试环境其实比这更为复杂，由于服务器上运行着其它的程序，对带宽的测试结果影响实际上超出了我的预期。这也是我的不足，即忽略了“这些因素的影响其实很大”，由于没有这一预设，我尝试了许多不同的优化方式，从 `pthread_barrier` 控制任务同时启动以缩小时间误差、调研了 NUMA 手动设置内存分配规则的方法，这还包括调研 NUMA 的优化手段（也算是补上了没认真听的课）。

对于并行编程，一直以来我没有尝试过自己写一些并行的代码，其一是找不到合适的练手项目，其二是不熟悉相关的接口。由于惯用的语言是 C++，因此在课程中使用 C++ 完成了要求的编程。在代码编写时，使用了一些 C++20 的并发支持的特性，如自动取消的线程 `std::jthread`、协作式取消请求 `std::stop_token`，除此之外，像信号量(semaphore)、闕(latch)与屏障(barrier)这样的同步原语，在 C++20 之前只能调用 C 语言的库函数，而今在 C++ 中也全部封装完好。实际上，C++ 实现并行的手段在 C++17 之后便丰富起来，像 `std::execution` 这样的并行调度器后续也在不断迭代。

3.2 建议

正如第 1 节指出：采用 O2 优化时，串行内存带宽测试不符合预期。建议考虑移除 O2 标记（可以在说明原因后再添加回来，以满足性能测试的需要）。

参考文献

- [1] Chen, Harry. “NUMA 处理器与进程绑定 - Harry Chen's Blog”[EB/OL]. (2022-05-08)[2025-01-14]. <https://harrychen.xyz/2022/05/08/numa-processor-and-cpu-binding/>.
- [2] Lemire, D. Estimating your memory bandwidth – Daniel Lemire's blog[EB/OL]. (2024-01-13)[2025-01-14]. <https://lemire.me/blog/2024/01/13/estimating-your-memory-bandwidth/>.
- [3] CPU 拓扑: 从 SMP 谈到 NUMA (理论篇)[EB/OL]. (2018-05-10)[2025-01-14]. <https://ctimbai.github.io/tags/NUMA/>
- [4] Hurd J. Verification of the Miller–Rabin probabilistic primality test[J]. The Journal of Logic and Algebraic Programming, 2003, 56(1-2): 3-21.
- [5] Bernstein D J. Distinguishing prime numbers from composite numbers: the state of the art in 2004[J]. URL: <http://cr.yp.to/papers.html#prime2004>, 2004, 23.
- [6] CPP reference: Concurrency support library[EB/OL]. [2024-11-8]. <https://en.cppreference.com/w/cpp/thread>
- [7] Peizhao Ou, Brian Demsky. Towards understanding the costs of avoiding out-of-thin-air results[J]. Proc. ACM Program. Lang., 2018, 2(OOPSLA): Article 136, 29 pages. DOI: <https://doi.org/10.1145/3276506>.