10244511412_P2

实践项目 P2：Profiling Serial Merge Sort

注：有一些数学公式在 gitea 上显示不是很好，建议阅读 pdf 版本

lscpu

记录 `lscpu` 的执行结果：

```
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   12
  On-line CPU(s) list:    0-11
Vendor ID:                GenuineIntel
  Model name:             12th Gen Intel(R) Core(TM) i5-12400F
    CPU family:           6
    Model:                151
    Thread(s) per core:   2
    Core(s) per socket:   6
    Socket(s):            1
    Stepping:             5
    CPU(s) scaling MHz:   19%
    CPU max MHz:          4400.0000
    CPU min MHz:          800.0000
    BogoMIPS:             4992.00
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
art arch_perfmon pebs bts rep_good nopl
                          xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq
dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
popcnt tsc_deadline_timer aes xsave av
                          x f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb ssbd ibrs
ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2
smep bmi2 erms invpcid rdseed adx smap cl
                          flushopt clwb intel_pt sha_ni xsaveopt xsavec xgetbv1 xsaves
split_lock_detect user_shstk avx_vnni dtherm ida arat pln pts hwp hwp_notify hwp_act_window
hwp_epp hwp_pkg_req hfi vnmi umip pku ospke waitp
                          kg gfni vaes vpclmulqdq rdpid movdiri movdir64b fsrm md_clear
serialize arch_lbr ibt flush_l1d arch_capabilities ibpb_exit_to_user
Virtualization features:
  Virtualization:         VT-x
Caches (sum of all):
  L1d:                    288 KiB (6 instances)
  L1i:                    192 KiB (6 instances)
  L2:                     7.5 MiB (6 instances)
  L3:                     18 MiB (1 instance)
NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-11
Vulnerabilities:
  Gather data sampling:   Not affected
  Itlb multihit:          Not affected
  L1tf:                   Not affected
  Mds:                    Not affected
  Meltdown:               Not affected
  Mmio stale data:        Not affected
```

```
  Reg file data sampling: Not affected
  Retbleed:               Not affected
  Spec rstack overflow:   Not affected
  Spec store bypass:      Mitigation; Speculative Store Bypass disabled via prctl
  Spectre v1:             Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:             Mitigation; Enhanced / Automatic IBRS; IBPB conditional; RSB filling;
PBRSB-eIBRS SW sequence; BHI BHI_DIS_S
  Srbds:                  Not affected
  Tsx async abort:        Not affected
  Vmscape:                Mitigation; IBPB before exit to userspace
```

Checkoff Item 1

安装 perf

我们先安装 perf：`sudo apt-get install linux-tools-common`，然后执行 `perf -v` 查看是否安装成功，但是却看到以下输出：

```
WARNING: perf not found for kernel 6.14.0-36
```

这似乎是 perf 与 `6.14.0-36-generic` 版本不兼容的一个 bug，该内核较新，我在网上也没找到好的解决方案，无奈尝试切换到 6.8 内核：

```
uname -r
perf -v
```

这下终于安装成功：

```
6.8.0-88-generic
perf version 6.8.12
```

找出性能瓶颈

我们先生成 `isort.c` 的可执行文件，根据作业指示，为了让 perf 能告诉我们具体是哪一行代码慢，需要开启 Debug 模式编译：

```
make isort DEBUG=1
```

然后运行插入排序程序，排序 10,000 个元素，重复 10 次，并让 perf 在后台记录 CPU 的行为：

```
perf record ./isort 10000 10
```

在当前目录下我们得到一个名为 `perf.data` 的文件，然后运行 `perf report` 来查看报告：

```
Samples: 880  of event 'cycles:P', Event count (approx.): 953452500
Overhead  Command  Shared Object         Symbol
  99.48%  isort    isort                 [.] isort
   0.23%  isort    libc.so.6             [.] rand_r
   0.11%  isort    isort                 [.] main
   0.07%  isort    isort                 [.] rand_r@plt
   0.05%  isort    [unknown]             [k] 0xffffffff9f605954
   0.03%  isort    [unknown]             [k] 0xffffffff9f6dbbb1
   0.01%  isort    ld-linux-x86-64.so.2  [.] _dl_sysdep_parse_arguments
   0.01%  isort    [unknown]             [k] 0xffffffff9fa69a5c
```

可以看到 `isort` 的负载是最高的，我们选择它然后进入 `Annotate isort`，得到：

```
       |    while (index >= left && *index > val) {
       |3b:   mov  -0x28(%rbp),%rcx
       |      xor  %eax,%eax
       |      cmp  -0x8(%rbp),%rcx
 20.07 |      mov  %al,-0x29(%rbp)
       |    ↓ jb   5d
       |      mov  -0x28(%rbp),%rax
       |      mov  (%rax),%eax
 23.62 |      cmp  -0x1c(%rbp),%eax
  0.23 |      seta %al
  0.80 |      mov  %al,-0x29(%rbp)
 23.71 |5d:   mov  -0x29(%rbp),%al
       |      test $0x1,%al
  2.65 |    ↓ jne  6d
       |    ↓ jmp  8b
       |    *(index + 1) = *index;
       |6d:   mov  -0x28(%rbp),%rax
  0.11 |      mov  (%rax),%ecx
       |      mov  -0x28(%rbp),%rax
 25.48 |      mov  %ecx,0x4(%rax)
       |    index--;
  0.23 |      mov  -0x28(%rbp),%rax
       |      add  $0xfffffffffffffffc,%rax
       |      mov  %rax,-0x28(%rbp)
       |    while (index >= left && *index > val) {
  3.10 |    ↑ jmp  3b
       |    }
```

我截取了显著开销很大的一部分，左侧的数字表示 CPU 采样周期占比，数字越大说明该行代码消耗的 CPU 资源越多。

其中最大的是几个 `cmp` 和 `mov` 操作，`mov %ecx,0x4` 对应代码：`*(index + 1) = *index;`，带来了 25.48% 的开销，这是插入排序中最核心的搬运步骤，当找到比插入值大的元素时，程序需要将其向后移动一位。这行代码涉及内存读取（`*index`）和内存写入（`*(index + 1)`），且位于内层 `while` 循环中，执行频率极高，因此占据了约 1/4 的总开销。

而其他的几个 `mov` 和 `cmp` 都是循环判断与比较带来的，占比大约 60% 多，对应代码：`while (index >= left && *index > val)`，每次循环迭代不仅要移动数据，还必须先从内存中读取当前元素与基准值 `val` 进行比较。频繁的内存访问和条件跳转，导致了大量的 CPU 周期消耗。

结论：所以该程序的核心瓶颈是插入排序的内层 `while` 循环以及里面涉及到的搬运操作。

Checkoff Item 2

安装 valgrind

安装 valgrind 并检查是否安装成功：

```
sudo apt-get install valgrind
valgrind --version
```

得到版本号，说明成功安装：

```
valgrind-3.22.0
```

分析缓存性能

先编译 `sum.c`

```
make sum
```

然后让 Valgrind 模拟并分析程序的缓存行为：

```
valgrind --tool=cachegrind --branch-sim=yes ./sum
```

得到：

```
==8776== Cachegrind, a high-precision tracing profiler
==8776== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==8776== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==8776== Command: ./sum
==8776==
Allocated array of size 10000000
Summing 100000000 random values...
Done. Value = 938895920
==8776==
==8776== I refs:        3,440,162,501
==8776==
==8776== Branches:        210,031,822  (110,031,437 cond + 100,000,385 ind)
==8776== Mispredicts:           4,179  (      4,000 cond +         179 ind)
==8776== Mispred rate:            0.0% (        0.0%     +         0.0%   )
```

我试了半天，尝试手动指定缓存大小，但一直没有我们想要的 L1 和 LLd 数据，这似乎是因为在新的版本的 valgrind 上默认不显示缓存数据，需要添加一个参数 `--cache-sim=yes`，这下终于有了：

```
==9425== Cachegrind, a high-precision tracing profiler
==9425== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==9425== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==9425== Command: ./sum
==9425==
--9425-- warning: L3 cache found, using its data for the LL simulation.
--9425-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--9425-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368
Allocated array of size 10000000
Summing 100000000 random values...
Done. Value = 938895920
==9425==
==9425== I refs:        3,440,162,501
==9425== I1  misses:            1,353
==9425== LLi misses:            1,346
==9425== I1  miss rate:          0.00%
==9425== LLi miss rate:          0.00%
==9425==
==9425== D refs:          610,050,465  (400,037,374 rd   + 210,013,091 wr)
==9425== D1  misses:      100,505,487  ( 99,880,046 rd   +     625,441 wr)
==9425== LLd misses:       53,572,509  ( 52,947,086 rd   +     625,423 wr)
==9425== D1  miss rate:          16.5% (       25.0%     +         0.3%  )
==9425== LLd miss rate:           8.8% (       13.2%     +         0.3%  )
==9425==
==9425== LL refs:         100,506,840  ( 99,881,399 rd   +     625,441 wr)
==9425== LL misses:        53,573,855  ( 52,948,432 rd   +     625,423 wr)
==9425== LL miss rate:            1.3% (        1.4%     +         0.3%  )
==9425==
==9425== Branches:        210,031,822  (110,031,437 cond + 100,000,385 ind)
==9425== Mispredicts:           4,179  (      4,000 cond +         179 ind)
==9425== Mispred rate:            0.0% (        0.0%     +         0.0%   )
```

我们先从 lscpu 中摘取出我们设备的缓存大小，然后开始分析：

```
Caches (sum of all):
  L1d:                    288 KiB (6 instances)
  L1i:                    192 KiB (6 instances)
  L2:                     7.5 MiB (6 instances)
  L3:                     18 MiB (1 instance)
```

程序 `sum.c` 分配了一个包含 10,000,000 个整数的数组，内存占用 $\approx 10,000,000 \times 4\ \text{Bytes} \approx \mathbf{40}\ \text{MB}$。同时，我们的 **L1d Cache:** 总共 288 KiB (6 instances)，即单核 $\approx \mathbf{48}\ \text{KB}$；**L3 Cache (LL):** 总共 $\mathbf{18}\ \text{MB}$。

L1 未命中数高达 100,505,487，这事实上是符合预期的，因为程序执行了 $N = 100,000,000$ 次循环，进行随机读取。由于 L1 缓存 (48 KB) 远小于数组大小 (40 MB)，且访问模式为随机，几乎每次访问新的数据索引都会导致 L1 未命中，造成了大量的未命中。

LLd 未命中数为 53,572,509 也是符合预期的，因为数组大小为 40 MB，而 L3 缓存为 18 MB，缓存能够容纳的数据比例约为 $18/40 = 45\%$，这意味着在完全随机访问的情况下，理论上有 $1 - 45\% = 55\%$ 的概率数据不在缓存中。而这样推出的理论值 $\approx 100,000,000 \times 55\% = 55,000,000$ 和实际测得的 53,572,509 非常接近，说明由于缓存容量不足加上随机访问模式，导致了大量的最后一级缓存未命中。

尝试降低缓存未命中率

我们先尝试降低 LLd 的未命中率，在之前的分析中，发现当 $U = 10,000,000$ 时，数组总大小约为 40 MB，由于我的 L3 缓存仅有 18 MB，大量数据无法驻留在缓存中，从而产生了高达 5300 万次的 `LLd misses`。

所以我们尝试将数组大小 U 从 $10,000,000$ 减少 为 $1,000,000$，使其能够被 L3 缓存完全容纳。新的数组大小为 $1,000,000 \times 4\ \text{Bytes} = \mathbf{4}\ \text{MB}$，由于 $\mathbf{4}\ \text{MB} < \mathbf{18}\ \text{MB}\ (\text{L3})$，所以理论上整个数组在第一次加载后应能常驻 L3 缓存，后续的随机访问应全部命中 L3。

```
实验结果
==10076==
==10076== I refs:        3,404,162,436
==10076== I1  misses:            1,353
==10076== LLi misses:            1,328
==10076== I1  miss rate:          0.00%
==10076== LLi miss rate:          0.00%
==10076==
==10076== D refs:          601,050,443  (400,037,358 rd   + 201,013,085 wr)
==10076== D1  misses:       98,838,890  ( 98,775,949 rd   +      62,941 wr)
==10076== LLd misses:           63,901  (      1,042 rd   +      62,859 wr)
==10076== D1  miss rate:          16.4% (       24.7%     +         0.0%  )
==10076== LLd miss rate:           0.0% (        0.0%     +         0.0%  )
==10076==
==10076== LL refs:          98,840,243  ( 98,777,302 rd   +      62,941 wr)
==10076== LL misses:            65,229  (      2,370 rd   +      62,859 wr)
==10076== LL miss rate:            0.0% (        0.0%     +         0.0%  )
==10076==
==10076== Branches:        201,031,810  (101,031,426 cond + 100,000,384 ind)
==10076== Mispredicts:           4,180  (      4,001 cond +         179 ind)
==10076== Mispred rate:            0.0% (        0.0%     +         0.0%   )
```

完全符合我们的理论分析，LLd Misses 为 `63,901`，相比之前的 5000 多万次，下降了 99.9/%，未命中率显示为 0.0/%（实际肯定不是 0，只是相较于其他未命中次数已经计算不出了）。结合 csapp 的知识，这里的 6 万次未命中应该主要是程序启动时的 Cold Misses 以及其他系统干扰，而后续绝大多数数据访问都成功在 L3 缓存中命中。

但是 L1 未命中数几乎没有下降多少，这也是正常的，尽管数组缩小了，但 4 MB 依然远大于 L1 缓存的 48 KB。

那么我猜测把 U 的大小降到很低应该能显著降低其未命中率，所以我把 `sum.c` 中的数组大小 U 修改为 4,000。这样它的占用内存大小只有 $4,000 \times 4\ \text{Bytes} = \mathbf{16}\ \text{KB}$，小于硬件的 48KB，所以理论上，数组完全可以被 L1 容纳。

```
==10464==
==10464== I refs:        3,400,178,169
==10464== I1  misses:            1,355
==10464== LLi misses:            1,329
==10464== I1  miss rate:          0.00%
==10464== LLi miss rate:          0.00%
==10464==
==10464== D refs:          600,054,364  (400,037,301 rd   + 200,017,063 wr)
==10464== D1  misses:            1,829  (      1,198 rd   +         631 wr)
==10464== LLd misses:            1,649  (      1,041 rd   +         608 wr)
==10464== D1  miss rate:           0.0% (        0.0%     +         0.0%  )
==10464== LLd miss rate:           0.0% (        0.0%     +         0.0%  )
==10464==
==10464== LL refs:               3,184  (      2,553 rd   +         631 wr)
==10464== LL misses:             2,978  (      2,370 rd   +         608 wr)
==10464== LL miss rate:            0.0% (        0.0%     +         0.0%  )
==10464==
==10464== Branches:        200,035,762  (100,035,382 cond + 100,000,380 ind)
==10464== Mispredicts:           4,191  (      4,012 cond +         179 ind)
==10464== Mispred rate:            0.0% (        0.0%     +         0.0%   )
```

这次 L1 未命中 仅有 1,829 次，考虑到程序总共进行了约 6 亿次数据引用（D refs），这仅有的 1,829 次未命中应该也来源于 Cold Misses。

上述两个实验验证了只要数据能塞进缓存里，即使是性能较差的随机访问也能获得极高的性能，因此减小工作集是降低未命中率的一个有效手段。

我们还应该 "Play with N"，但是 N 只决定循环次数，对未命中率应该没多少影响，只会改变它的数值大小，而且应该呈线性关系。我们把 U 调回原始的 1千万，将循环次数 N 从原始的 $100,000,000$ (1亿) 减少至 $10,000,000$ (1千万)，即减少 10 倍。

```
==10776==
==10776== I refs:        380,162,501
==10776== I1  misses:          1,353
==10776== LLi misses:          1,346
==10776== I1  miss rate:        0.00%
==10776== LLi miss rate:        0.00%
==10776==
==10776== D refs:         70,050,465  (40,037,374 rd   + 30,013,091 wr)
==10776== D1  misses:     10,614,505  ( 9,989,064 rd   +    625,441 wr)
==10776== LLd misses:      5,920,966  ( 5,295,543 rd   +    625,423 wr)
==10776== D1  miss rate:        15.2% (      24.9%     +        2.1%  )
==10776== LLd miss rate:         8.5% (      13.2%     +        2.1%  )
==10776==
==10776== LL refs:        10,615,858  ( 9,990,417 rd   +    625,441 wr)
==10776== LL misses:       5,922,312  ( 5,296,889 rd   +    625,423 wr)
==10776== LL miss rate:          1.3% (       1.3%     +        2.1%  )
==10776==
==10776== Branches:       30,031,822  (20,031,437 cond + 10,000,385 ind)
==10776== Mispredicts:         4,179  (     4,000 cond +        179 ind)
==10776== Mispred rate:          0.0% (       0.0%     +        0.0%   )
```

结果跟我们的分析也高度一致，未命中总数随 N 的减少呈现出完美的线性关系（均减少了约 10 倍）。这是因为程序执行的总内存访问次数减少了，自然遭遇未命中的绝对次数也随之减少。同时尽管 N 发生了剧烈变化，但未命中率几乎保持不变。这说明对于随机访问的大数组，其缓存命中的概率（概率由 U 和缓存大小之比决定）是一个固有属性，与我们采样多少次 (N) 无关。

总的来说，要真正优化程序的缓存性能，我们的工作重心应该放在减小工作集 U 上，比如对数据进行分块。但其实我感觉更好的方法还是优化访问模式，但条件受限，没法继续进行实验探索了。

Write-up 1

对比 DEBUG = 0 和 DEBUG = 1

先查看在 `DEBUG=1` 下的 Cachegrind 输出：

```
make clean
make DEBUG=1
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes ./sort 10000 10
```

得到输出：

```
==13508== Cachegrind, a high-precision tracing profiler
==13508== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==13508== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==13508== Command: ./sort 10000 10
==13508==
--13508-- warning: L3 cache found, using its data for the LL simulation.
--13508-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--13508-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.030110 sec
sort_a repeated : Elapsed execution time: 0.029734 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.059154 sec
sort_a repeated : Elapsed execution time: 0.058719 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==13508==
==13508== I refs:        416,875,191
==13508== I1  misses:          1,699
==13508== LLi misses:          1,604
==13508== I1  miss rate:        0.00%
==13508== LLi miss rate:        0.00%
==13508==
==13508== D refs:        268,508,582  (197,902,983 rd   + 70,605,599 wr)
==13508== D1  misses:        225,938  (    124,983 rd   +    100,955 wr)
==13508== LLd misses:          3,599  (      1,275 rd   +      2,324 wr)
==13508== D1  miss rate:         0.1% (        0.1%     +        0.1%  )
==13508== LLd miss rate:         0.0% (        0.0%     +        0.0%  )
==13508==
==13508== LL refs:           227,637  (    126,682 rd   +    100,955 wr)
==13508== LL misses:           5,203  (      2,879 rd   +      2,324 wr)
==13508== LL miss rate:          0.0% (        0.0%     +        0.0%  )
==13508==
==13508== Branches:       41,164,175  ( 39,463,491 cond +  1,700,684 ind)
==13508== Mispredicts:     3,457,958  (  3,457,674 cond +        284 ind)
==13508== Mispred rate:          8.4% (        8.8%     +        0.0%   )
```

然后编译 `DEBUG=0`：

```
make clean
make DEBUG=0
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes ./sort 10000 10
```

得到：

```
==15337== Cachegrind, a high-precision tracing profiler
==15337== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==15337== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==15337== Command: ./sort 10000 10
==15337==
--15337-- warning: L3 cache found, using its data for the LL simulation.
--15337-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--15337-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.015753 sec
sort_a repeated : Elapsed execution time: 0.015329 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.031255 sec
sort_a repeated : Elapsed execution time: 0.030790 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==15337==
==15337== I refs:        251,381,470
==15337== I1  misses:          1,736
==15337== LLi misses:          1,639
==15337== I1  miss rate:        0.00%
==15337== LLi miss rate:        0.00%
==15337==
==15337== D refs:         92,284,481  (54,849,834 rd   + 37,434,647 wr)
==15337== D1  misses:        226,518  (   125,232 rd   +    101,286 wr)
==15337== LLd misses:          3,615  (     1,290 rd   +      2,325 wr)
==15337== D1  miss rate:         0.2% (       0.2%     +        0.3%  )
==15337== LLd miss rate:         0.0% (       0.0%     +        0.0%  )
==15337==
==15337== LL refs:           228,254  (   126,968 rd   +    101,286 wr)
==15337== LL misses:           5,254  (     2,929 rd   +      2,325 wr)
==15337== LL miss rate:          0.0% (       0.0%     +        0.0%  )
==15337==
==15337== Branches:       28,835,901  (27,135,187 cond +  1,700,714 ind)
==15337== Mispredicts:     1,103,173  ( 1,102,868 cond +        305 ind)
==15337== Mispred rate:          3.8% (       4.1%     +        0.0%   )
```

性能指标 (Metric)	DEBUG=0 (Optimized)	DEBUG=1 (Unoptimized)	差异幅度 (Delta)
指令引用 (I refs)	2,783,693,443	4,676,551,362	-40.5%

性能指标 (Metric)	DEBUG=0 (Optimized)	DEBUG=1 (Unoptimized)	差异幅度 (Delta)
数据引用 (D refs)	1,015,802,272	3,072,540,949	-66.9%
分支指令 (Branches)	305,588,847	459,057,080	-33.4%
分支误预测率 (Mispred rate)	3.9%	7.7%	-3.8%

开启优化后，指令总数减少了约 **40%**，这是因为编译器执行了内联、循环展开和死代码消除等优化，去除了调试版本中大量的冗余指令。数据访问次数的下降更为剧烈，减少了约 **65%**，这可能是因为优化器极大地改善了寄存器分配，在调试模式下，变量通常被强制保存在栈内存中以便调试器读取；而在优化模式下，频繁使用的变量被保存在 CPU 寄存器中，避免了昂贵的内存读写操作。同时优化后的代码不仅分支指令更少，而且误预测率也降低了一半以上，最终体现为运行速度提升了一半左右。

Advantages and Disadvantages of Instruction Count as a Metric

优点

1. **确定性与可复现性:**
   指令数是一个确定的指标。对于相同的输入，无论在何时运行、无论机器负载如何，程序的指令数通常保持不变。相比之下，运行时间会受到操作系统调度和后台进程等的严重干扰，导致测量结果波动。
2. **硬件频率无关性:**
   指令数屏蔽了 CPU 时钟频率的影响（如动态调频、睿频等）。它更能反映算法本身的逻辑复杂度以及编译器生成代码的效率。

缺点

1. **忽略指令延迟差异:**
   并非所有指令的执行成本都是相等的。一条简单的整数加法指令可能只需要 1 个时钟周期，而一条导致缓存未命中的内存读取指令可能需要数百个周期。如果优化减少了总指令数，但增加了昂贵指令（如除法或内存访问）的比例，程序反而可能变慢。
2. **无法反映存储器层次结构性能:**
   仅看 `I refs` 无法揭示缓存局部性，两个指令数相同的程序，如果一个频繁发生 L1/L2 Cache Miss，其速度会远远慢于另一个。体现在我们的实验数据中，虽然优化版指令数大减，但 `D1 misses` 数量相差不大（约 22.6k）。
3. **忽略指令级并行:**
   现代 CPU 是超标量架构，可以同时发射和执行多条指令。如果代码虽然指令数多，但指令间依赖性低，能够充分利用流水线并行，其运行速度可能会超过指令数少但存在严重数据依赖（导致 Pipeline Stall）的代码。

Write-up 2

注：Write-up 2,3 均是在完成必做题后做的

选择内联的函数

我选择内联 `copy_i` 和 `merge_i`，因为 `copy_i` 是一个非常短小的函数，仅包含一个简单的循环，函数调用的开销（压栈、跳转、返回）相对于函数体执行的时间来说占比较大，非常适合内联；而 `merge_i` 是归并排序的核心操作，在排序过程中被频繁调用，将其内联可以减少从 `sort_i` 调用它的开销。

然后编写代码：

```
#include "./util.h"

static inline void merge_i(data_t* A, int p, int q, int r);
static inline void copy_i(data_t* source, data_t* dest, int n);

void sort_i(data_t* A, int p, int r) {
  assert(A);
  if (p < r) {
    int q = (p + r) / 2;
    sort_i(A, p, q);
```

```
    sort_i(A, q + 1, r);
    merge_i(A, p, q, r);
  }
}

static inline void merge_i(data_t* A, int p, int q, int r) {
  assert(A);
  assert(p <= q);
  assert((q + 1) <= r);
  int n1 = q - p + 1;
  int n2 = r - q;

  data_t* left = 0, * right = 0;
  mem_alloc(&left, n1 + 1);
  mem_alloc(&right, n2 + 1);
  if (left == NULL || right == NULL) {
    mem_free(&left);
    mem_free(&right);
    return;
  }

  copy_i(&(A[p]), left, n1);
  copy_i(&(A[q + 1]), right, n2);
  left[n1] = UINT_MAX;
  right[n2] = UINT_MAX;

  int i = 0;
  int j = 0;

  for (int k = p; k <= r; k++) {
    if (left[i] <= right[j]) {
      A[k] = left[i];
      i++;
    } else {
      A[k] = right[j];
      j++;
    }
  }
  mem_free(&left);
  mem_free(&right);
}

static inline void copy_i(data_t* source, data_t* dest, int n) {
  assert(dest);
  assert(source);

  for (int i = 0 ; i < n ; i++) {
    dest[i] = source[i];
  }
}
```

性能分析

我们照例将其注册然后测试：

```
make clean
make sort
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes ./sort 400000 10
```

`sort_i.c`：

```
==40092== Cachegrind, a high-precision tracing profiler
==40092== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==40092== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==40092== Command: ./sort 400000 10
==40092==
--40092-- warning: L3 cache found, using its data for the LL simulation.
--40092-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--40092-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_i          : Elapsed execution time: 0.703386 sec
Generating inverted array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_i          : Elapsed execution time: 1.405519 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==40092==
==40092== I refs:        5,675,458,821
==40092== I1  misses:            1,766
==40092== LLi misses:            1,657
==40092== I1  miss rate:          0.00%
==40092== LLi miss rate:          0.00%
==40092==
==40092== D refs:        1,971,076,333  (1,180,356,567 rd   + 790,719,766 wr)
==40092== D1  misses:       15,965,905  (    8,095,970 rd   +   7,869,935 wr)
==40092== LLd misses:          101,779  (        1,287 rd   +     100,492 wr)
==40092== D1  miss rate:           0.8% (          0.7%     +         1.0%  )
==40092== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==40092==
==40092== LL refs:          15,967,671  (    8,097,736 rd   +   7,869,935 wr)
==40092== LL misses:           103,436  (        2,944 rd   +     100,492 wr)
==40092== LL miss rate:            0.0% (          0.0%     +         0.0%  )
==40092==
==40092== Branches:        591,417,968  (  555,417,266 cond +  36,000,702 ind)
==40092== Mispredicts:      26,712,325  (   26,712,026 cond +         299 ind)
==40092== Mispred rate:            4.5% (          4.8%     +         0.0%   )
```

然后贴一下实际是在 write-up 4 测出的 `sort_a.c` 数据：

```
==17677== Cachegrind, a high-precision tracing profiler
==17677== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==17677== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==17677== Command: ./sort 400000 10
==17677==
--17677-- warning: L3 cache found, using its data for the LL simulation.
--17677-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--17677-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
```

```
sort_a          : Elapsed execution time: 0.700789 sec
Generating inverted array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 1.396984 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==17677==
==17677== I refs:        5,675,458,830
==17677== I1  misses:            1,762
==17677== LLi misses:            1,656
==17677== I1  miss rate:          0.00%
==17677== LLi miss rate:          0.00%
==17677==
==17677== D refs:        1,971,076,332  (1,180,356,565 rd   + 790,719,767 wr)
==17677== D1  misses:       15,965,905  (    8,095,970 rd   +   7,869,935 wr)
==17677== LLd misses:          101,779  (        1,287 rd   +     100,492 wr)
==17677== D1  miss rate:           0.8% (          0.7%     +         1.0%  )
==17677== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==17677==
==17677== LL refs:          15,967,667  (    8,097,732 rd   +   7,869,935 wr)
==17677== LL misses:           103,435  (        2,943 rd   +     100,492 wr)
==17677== LL miss rate:            0.0% (          0.0%     +         0.0%  )
==17677==
==17677== Branches:        591,417,972  (  555,417,270 cond +  36,000,702 ind)
==17677== Mispredicts:      24,633,613  (   24,633,314 cond +         299 ind)
==17677== Mispred rate:            4.2% (          4.4%     +         0.0%   )
```

列表：

Metric	sort_a	sort_i	Delta
随机数组耗时	0.700789 sec	0.703386 sec	+0.3%
倒序数组耗时	1.396984 sec	1.405519 sec	+0.6%
总指令数 (I refs)	5,675,458,830	5,675,458,821	-9 条指令
数据引用 (D refs)	1,971,076,332	1,971,076,333	+1
L1 数据缓存未命中 (D1 misses)	15,965,905	15,965,905	0
分支指令数 (Branches)	591,417,972	591,417,968	-4
分支预测错误 (Mispredicts)	24,633,613	26,712,325	+8.4%

可以说是完全没有区别了，在执行了数十亿条指令的测试中，指令数差异仅为 9 条，这说明两者的二进制机器码几乎完全相同。

我估计是编译器优化等级太高了，忽略了我的 `inline` 建议。在现代编译器中，`inline` 关键字并非强制指令。编译器可能基于自身的成本收益分析模型，判断内联 `merge_i` 不会带来收益，或者会导致代码膨胀，因此拒绝了内联。

Write-up 3

实在没时间做实验了，就从理论上回答一下 write-up 3 的问题。

内联递归函数可能带来的性能负面影响

内联一个递归函数要求编译器对递归进行“展开”，即在转为普通函数调用之前，将函数体复制固定次数。由于归并排序算法通常在每一层递归中进行两次调用，这种展开会导致代码体积随着展开层数的增加呈指数级增长，也就是代码膨胀。

这样会增加程序的指令工作集，如果这个工作集的大小超过了 CPU L1 指令缓存的话，CPU 就无法将频繁执行的指令保持在最快的存储器中；同时递归入口附近的控制流变得更加复杂，增加了 branch predictor 的负担。

另外，递归函数无法被无限 inline，编译器通常只会展开前几层，深层递归仍然需要正常调用，这意味着收益有限而开销更大。

Cachegrind 分析数据如何帮助衡量这些负面影响

结合上述的分析，要验证内联是否因代码膨胀导致了性能下降，我们可以查看 Cachegrind 输出中的以下几个指标：

- **I1 misses / I1 miss rate**：判断指令缓存是否因代码体积变大而出现溢出。
- **LLi misses**：观察更高层次指令缓存是否也受到了污染。
- **Instructions executed**：检测是否因为内联复制了更多指令。
- **Branch misses（Bc-misses）**：判断控制流复杂化后，分支预测是否变差。

如果这些指标在 inline 递归版本中总体上更高，就可以认为内联递归确实带来了负面影响。

Write-up 4

改写指针访问版本

注：由于我是先完成必做作业再完成选做的 write-up 2,3,7，所以下面修改的代码是基于 `sort_a.c` 修改的，write-up 4,5,6 均不包含 inlining 优化

我们先根据作业要求完成指针访问的版本的 `sort_p.c`：

```c
#include "./util.h"

static void merge_p(data_t* A, int p, int q, int r);
static void copy_p(data_t* source, data_t* dest, int n);

void sort_p(data_t* A, int p, int r) {
  assert(A);
  if (p < r) {
    int q = (p + r) / 2;
    sort_p(A, p, q);
    sort_p(A, q + 1, r);
    merge_p(A, p, q, r);
  }
}

static void merge_p(data_t* A, int p, int q, int r) {
  assert(A);
  assert(p <= q);
  assert((q + 1) <= r);
  int n1 = q - p + 1;
  int n2 = r - q;

  data_t* left = 0, * right = 0;
  mem_alloc(&left, n1 + 1);
  mem_alloc(&right, n2 + 1);
  if (left == NULL || right == NULL) {
    mem_free(&left);
    mem_free(&right);
    return;
  }
```

```
  copy_p(A + p, left, n1);
  copy_p(A + q + 1, right, n2);

  *(left + n1) = UINT_MAX;
  *(right + n2) = UINT_MAX;

  data_t *l_ptr = left;
  data_t *r_ptr = right;
  data_t *dest_ptr = A + p;

  for (int k = p; k <= r; k++) {
    if (*l_ptr <= *r_ptr) {
      *dest_ptr = *l_ptr;
      l_ptr++;
    } else {
      *dest_ptr = *r_ptr;
      r_ptr++;
    }
    dest_ptr++;
  }

  mem_free(&left);
  mem_free(&right);
}

static void copy_p(data_t* source, data_t* dest, int n) {
  assert(dest);
  assert(source);

  data_t* source_end = source + n;

  while (source < source_end) {
    *dest++ = *source++;
  }
}
```

进行性能分析

我们先在 `main.c` 里取消注释 `{&sort_p, "sort_p\t\t"},`，然后运行：

```
make clean
make sort
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes ./sort 10000 10
```

得到：

```
==17094== Cachegrind, a high-precision tracing profiler
==17094== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==17094== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==17094== Command: ./sort 10000 10
==17094==
--17094-- warning: L3 cache found, using its data for the LL simulation.
--17094-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--17094-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
```

```
sort_p          : Elapsed execution time: 0.015506 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_p          : Elapsed execution time: 0.030589 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==17094==
==17094== I refs:        127,624,624
==17094== I1  misses:          1,732
==17094== LLi misses:          1,634
==17094== I1  miss rate:        0.00%
==17094== LLi miss rate:        0.00%
==17094==
==17094== D refs:         47,290,260  (28,001,185 rd   + 19,289,075 wr)
==17094== D1  misses:        130,098  (    68,097 rd   +     62,001 wr)
==17094== LLd misses:          3,614  (     1,289 rd   +      2,325 wr)
==17094== D1  miss rate:         0.3% (       0.2%     +        0.3%  )
==17094== LLd miss rate:         0.0% (       0.0%     +        0.0%  )
==17094==
==17094== LL refs:           131,830  (    69,829 rd   +     62,001 wr)
==17094== LL misses:           5,248  (     2,923 rd   +      2,325 wr)
==17094== LL miss rate:          0.0% (       0.0%     +        0.0%  )
==17094==
==17094== Branches:       14,944,661  (14,043,961 cond +    900,700 ind)
==17094== Mispredicts:       579,349  (   579,050 cond +        299 ind)
==17094== Mispred rate:          3.9% (       4.1%     +        0.0%   )
```

为了对比，我们需要在非 DEBUG 模式下跑一下 `sort_a.c`，得到以下输出：

```
==17371== Cachegrind, a high-precision tracing profiler
==17371== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==17371== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==17371== Command: ./sort 10000 10
==17371==
--17371-- warning: L3 cache found, using its data for the LL simulation.
--17371-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--17371-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.015835 sec
Generating inverted array of 10000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.031241 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==17371==
```

```
==17371== I refs:        127,624,560
==17371== I1  misses:          1,728
==17371== LLi misses:          1,632
==17371== I1  miss rate:        0.00%
==17371== LLi miss rate:        0.00%
==17371==
==17371== D refs:         47,290,240  (28,001,167 rd   + 19,289,073 wr)
==17371== D1  misses:        130,098  (    68,097 rd   +     62,001 wr)
==17371== LLd misses:          3,614  (     1,289 rd   +      2,325 wr)
==17371== D1  miss rate:         0.3% (       0.2%     +        0.3%  )
==17371== LLd miss rate:         0.0% (       0.0%     +        0.0%  )
==17371==
==17371== LL refs:           131,826  (    69,825 rd   +     62,001 wr)
==17371== LL misses:           5,246  (     2,921 rd   +      2,325 wr)
==17371== LL miss rate:          0.0% (       0.0%     +        0.0%  )
==17371==
==17371== Branches:       14,944,645  (14,043,945 cond +    900,700 ind)
==17371== Mispredicts:       554,566  (   554,267 cond +        299 ind)
==17371== Mispred rate:          3.7% (       3.9%     +        0.0%   )
```

我发现这几乎完全一样，还以为是哪里运行错了，反复测了几遍结果也一样，有可能是数据太小了，我把数据调成 400000 重新测试：

`sort_a.c`：

```
==17677== Cachegrind, a high-precision tracing profiler
==17677== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==17677== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==17677== Command: ./sort 400000 10
==17677==
--17677-- warning: L3 cache found, using its data for the LL simulation.
--17677-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--17677-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 0.700789 sec
Generating inverted array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_a          : Elapsed execution time: 1.396984 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==17677==
==17677== I refs:        5,675,458,830
==17677== I1  misses:            1,762
==17677== LLi misses:            1,656
==17677== I1  miss rate:          0.00%
==17677== LLi miss rate:          0.00%
==17677==
==17677== D refs:        1,971,076,332  (1,180,356,565 rd   + 790,719,767 wr)
==17677== D1  misses:       15,965,905  (    8,095,970 rd   +   7,869,935 wr)
==17677== LLd misses:          101,779  (        1,287 rd   +     100,492 wr)
==17677== D1  miss rate:           0.8% (          0.7%     +         1.0%  )
```

```
==17677== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==17677==
==17677== LL refs:          15,967,667  (    8,097,732 rd   +   7,869,935 wr)
==17677== LL misses:           103,435  (        2,943 rd   +     100,492 wr)
==17677== LL miss rate:            0.0% (          0.0%     +         0.0%  )
==17677==
==17677== Branches:        591,417,972  (  555,417,270 cond +  36,000,702 ind)
==17677== Mispredicts:      24,633,613  (   24,633,314 cond +         299 ind)
==17677== Mispred rate:            4.2% (          4.4%     +         0.0%   )
```

`sort_p.c`：

```
==17831== Cachegrind, a high-precision tracing profiler
==17831== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==17831== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==17831== Command: ./sort 400000 10
==17831==
--17831-- warning: L3 cache found, using its data for the LL simulation.
--17831-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--17831-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_p          : Elapsed execution time: 0.679514 sec
Generating inverted array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_p          : Elapsed execution time: 1.355971 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==17831==
==17831== I refs:        5,675,458,843
==17831== I1  misses:            1,778
==17831== LLi misses:            1,656
==17831== I1  miss rate:          0.00%
==17831== LLi miss rate:          0.00%
==17831==
==17831== D refs:        1,971,076,336  (1,180,356,569 rd   + 790,719,767 wr)
==17831== D1  misses:       15,965,905  (    8,095,970 rd   +   7,869,935 wr)
==17831== LLd misses:          101,779  (        1,287 rd   +     100,492 wr)
==17831== D1  miss rate:           0.8% (          0.7%     +         1.0%  )
==17831== LLd miss rate:           0.0% (          0.0%     +         0.0%  )
==17831==
==17831== LL refs:          15,967,683  (    8,097,748 rd   +   7,869,935 wr)
==17831== LL misses:           103,435  (        2,943 rd   +     100,492 wr)
==17831== LL miss rate:            0.0% (          0.0%     +         0.0%  )
==17831==
==17831== Branches:        591,417,977  (  555,417,275 cond +  36,000,702 ind)
==17831== Mispredicts:      26,789,125  (   26,788,826 cond +         299 ind)
==17831== Mispred rate:            4.5% (          4.8%     +         0.0%   )
```

可以看到虽然时间略有提升，但是缓存未命中还是几乎一样，看在优化下指针版本和数组版本确实没有太大的性能差异。主要原因还是现代编译器非常智能，能够自动检测到循环中的数组索引模式，并在生成机器码时自动将其转换为等

效的高效指针算术逻辑。因此，无论源码写的是数组还是指针，最终生成的汇编代码是高度相似甚至一样的。

我们来反汇编验证一下 `objdump -d -S sort > sort.asm`：

`sort_a.c`：

```
    if (left[i] <= right[j]) {
    2aa0:   45 89 ca                mov    %r9d,%r10d
    2aa3:   46 8b 1c 90             mov    (%rax,%r10,4),%r11d
    2aa7:   41 89 f2                mov    %esi,%r10d
    2aaa:   42 8b 2c 91             mov    (%rcx,%r10,4),%ebp
    2aae:   45 31 d2                xor    %r10d,%r10d
    2ab1:   45 31 f6                xor    %r14d,%r14d
    2ab4:   41 39 eb                cmp    %ebp,%r11d
    2ab7:   41 0f 97 c2             seta   %r10b
    2abb:   41 0f 96 c6             setbe  %r14b
    2abf:   41 0f 42 eb             cmovb  %r11d,%ebp
    2ac3:   45 01 ce                add    %r9d,%r14d
    2ac6:   41 01 f2                add    %esi,%r10d
    2ac9:   89 2c 93                mov    %ebp,(%rbx,%rdx,4)
    2acc:   46 8b 1c b0             mov    (%rax,%r14,4),%r11d
    2ad0:   42 8b 2c 91             mov    (%rcx,%r10,4),%ebp
    2ad4:   31 f6                   xor    %esi,%esi
    2ad6:   45 31 c9                xor    %r9d,%r9d
    2ad9:   41 39 eb                cmp    %ebp,%r11d
    2adc:   40 0f 97 c6             seta   %sil
    2ae0:   41 0f 96 c1             setbe  %r9b
    2ae4:   41 0f 42 eb             cmovb  %r11d,%ebp
    2ae8:   45 01 f1                add    %r14d,%r9d
    2aeb:   44 01 d6                add    %r10d,%esi
    2aee:   89 6c 93 04             mov    %ebp,0x4(%rbx,%rdx,4)
  for (int k = p; k <= r; k++) {
    2af2:   48 83 c2 02             add    $0x2,%rdx
    2af6:   41 83 c0 fe             add    $0xfffffffe,%r8d
    2afa:   75 a4                   jne    2aa0 <sort_a+0x2b0>
```

`sort_p.c`：

```
    if (left[i] <= right[j]) {
    2e10:   45 89 ca                mov    %r9d,%r10d
    2e13:   46 8b 1c 90             mov    (%rax,%r10,4),%r11d
    2e17:   41 89 f2                mov    %esi,%r10d
    2e1a:   42 8b 2c 91             mov    (%rcx,%r10,4),%ebp
    2e1e:   45 31 d2                xor    %r10d,%r10d
    2e21:   45 31 f6                xor    %r14d,%r14d
    2e24:   41 39 eb                cmp    %ebp,%r11d
    2e27:   41 0f 97 c2             seta   %r10b
    2e2b:   41 0f 96 c6             setbe  %r14b
    2e2f:   41 0f 42 eb             cmovb  %r11d,%ebp
    2e33:   45 01 ce                add    %r9d,%r14d
    2e36:   41 01 f2                add    %esi,%r10d
    2e39:   89 2c 93                mov    %ebp,(%rbx,%rdx,4)
    2e3c:   46 8b 1c b0             mov    (%rax,%r14,4),%r11d
    2e40:   42 8b 2c 91             mov    (%rcx,%r10,4),%ebp
    2e44:   31 f6                   xor    %esi,%esi
    2e46:   45 31 c9                xor    %r9d,%r9d
    2e49:   41 39 eb                cmp    %ebp,%r11d
    2e4c:   40 0f 97 c6             seta   %sil
    2e50:   41 0f 96 c1             setbe  %r9b
    2e54:   41 0f 42 eb             cmovb  %r11d,%ebp
    2e58:   45 01 f1                add    %r14d,%r9d
```

```
    2e5b:   44 01 d6                add    %r10d,%esi
    2e5e:   89 6c 93 04             mov    %ebp,0x4(%rbx,%rdx,4)
  for (int k = p; k <= r; k++) {
    2e62:   48 83 c2 02             add    $0x2,%rdx
    2e66:   41 83 c0 fe             add    $0xfffffffe,%r8d
    2e6a:   75 a4                   jne    2e10 <sort_p+0x2b0>
```

可以看到几乎没有任何差别，符合我们之前的猜想。那为什么 `sort_p` 还是比 `sort_a` 稍微快一点点呢?

我尝试理论分析了一下，数组索引 (`A[i]`) 是一个二元运算，为了获取内存地址，CPU 要执行线性变换公式：

$$\text{Address} = \text{Base\_Addr} + (\text{Index} \times \text{Element\_Size})$$

这意味着每次访问都需要进行一次乘法（或移位）和一次加法运算。此外，还需要额外维护 `Index` 变量的自增。

而指针算术 (`*ptr++`) 是一个一元更新，访问下一个元素仅需执行简单的增量操作：

$$\text{New\_Addr} = \text{Current\_Addr} + \text{Element\_Size}$$

这消除了“基地址”和“偏移量”的概念，将复杂的地址计算简化为单一的加法指令。

结合寄存器知识的话，数组在循环中，为了定位元素，CPU 理论上需要同时维护两个状态：基地址和索引计数器，这至少占用了两个寄存器。而指针版只需要维护当前指针，减少了对寄存器的需求。在复杂的递归函数（如归并排序）中，节省下来的寄存器可以用于存储更多的临时变量。

当然这有点为了解释而解释，实验中两者是差不多的。

Write-up 5

用冒泡排序来排序 base case

在指针版本的基础上，我采用冒泡排序来作为 base case 的排序算法，同时设置了一个 threshold，默认值是 16:

```c
#include "./util.h"

#define COARSEN_THRESHOLD 16

static void merge_c(data_t* A, int p, int q, int r);
static void copy_c(data_t* source, data_t* dest, int n);
static void bubble_sort(data_t* A, int p, int r);

void sort_c(data_t* A, int p, int r) {
  assert(A);

  int len = r - p + 1;

  if (len <= COARSEN_THRESHOLD) {
      bubble_sort(A, p, r);
  }
  else if (p < r) {
    int q = (p + r) / 2;
    sort_c(A, p, q);
    sort_c(A, q + 1, r);
    merge_c(A, p, q, r);
  }
}

static void bubble_sort(data_t* A, int p, int r) {
    int len = r - p + 1;
    for (int i = 0; i < len - 1; i++) {
        for (int j = p; j < r - i; j++) {
            if (A[j] > A[j+1]) {
                data_t temp = A[j];
```

```
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}

static void merge_c(data_t* A, int p, int q, int r) {
  assert(A);
  assert(p <= q);
  assert((q + 1) <= r);
  int n1 = q - p + 1;
  int n2 = r - q;

  data_t* left = 0, * right = 0;
  mem_alloc(&left, n1 + 1);
  mem_alloc(&right, n2 + 1);
  if (left == NULL || right == NULL) {
    mem_free(&left);
    mem_free(&right);
    return;
  }

  copy_c(A + p, left, n1);
  copy_c(A + q + 1, right, n2);

  *(left + n1) = UINT_MAX;
  *(right + n2) = UINT_MAX;

  data_t *l_ptr = left;
  data_t *r_ptr = right;
  data_t *dest_ptr = A + p;

  for (int k = p; k <= r; k++) {
    if (*l_ptr <= *r_ptr) {
      *dest_ptr = *l_ptr;
      l_ptr++;
    } else {
      *dest_ptr = *r_ptr;
      r_ptr++;
    }
    dest_ptr++;
  }

  mem_free(&left);
  mem_free(&right);
}

static void copy_c(data_t* source, data_t* dest, int n) {
  assert(dest);
  assert(source);

  data_t* source_end = source + n;
  while (source < source_end) {
    *dest++ = *source++;
  }
}
```

初步性能分析

我们先在 threshold 为 16 时做一下性能分析：

```
make clean
make sort
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes ./sort 400000 10
```

数据：

```
==19318== Cachegrind, a high-precision tracing profiler
==19318== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==19318== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==19318== Command: ./sort 400000 10
==19318==
--19318-- warning: L3 cache found, using its data for the LL simulation.
--19318-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--19318-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_c          : Elapsed execution time: 0.267813 sec
Generating inverted array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_c          : Elapsed execution time: 0.522963 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==19318==
==19318== I refs:        2,563,812,419
==19318== I1  misses:            1,776
==19318== LLi misses:            1,651
==19318== I1  miss rate:          0.00%
==19318== LLi miss rate:          0.00%
==19318==
==19318== D refs:          720,959,502  (430,163,853 rd   + 290,795,649 wr)
==19318== D1  misses:       15,943,300  (  8,084,039 rd   +   7,859,261 wr)
==19318== LLd misses:          101,777  (      1,287 rd   +     100,490 wr)
==19318== D1  miss rate:           2.2% (        1.9%     +         2.7%  )
==19318== LLd miss rate:           0.0% (        0.0%     +         0.0%  )
==19318==
==19318== LL refs:          15,945,076  (  8,085,815 rd   +   7,859,261 wr)
==19318== LL misses:           103,428  (      2,938 rd   +     100,490 wr)
==19318== LL miss rate:            0.0% (        0.0%     +         0.0%  )
==19318==
==19318== Branches:        250,203,844  (243,581,702 cond +   6,622,142 ind)
==19318== Mispredicts:      19,032,537  ( 19,032,238 cond +         299 ind)
==19318== Mispred rate:            7.6% (        7.8%     +         0.0%   )
```

我们对比一下之前 `sort_p` 的性能：

性能指标 (Metric)	sort_p (Pointer)	sort_c (Coarsening)	性能提升 (Improvement)
I refs (Instructions)	**5,675,458,843**	**2,563,812,419**	**-54.8%** (指令数减半)
D refs (Data Refs)	**1,971,076,336**	**720,959,502**	**-63.4%** (访存大幅减少)
Branches	**591,417,977**	**250,203,844**	**-57.7%**

性能指标 (Metric)	sort_p (Pointer)	sort_c (Coarsening)	性能提升 (Improvement)
Execution Time	**0.679514 sec**	**0.267813 sec**	**2.54x 加速**

性能得到了非常显著的提升，我们来分析一下原因。首先，在 `sort_p` 中，每一层递归的 `merge` 函数都会调用 `malloc` 和 `free` 来创建临时数组，在数据量极大时，归并排序的递归树底部节点数量巨大，通过在 $N = 16$ 时停止递归，我们直接剪除了递归树最底部的 4 层，避免了大量的的 `malloc` / `free` 系统调用。

我还注意到 D refs 减少了约 63%，这说明大量的 CPU 时间原本被浪费在维护递归栈帧、参数传递以及堆内存管理的元数据操作上，而不是实际的排序比较上，而base case 换成冒泡排序可以大幅减少这些开销。

这样做还带来了更好的缓存局部性，冒泡排序在处理小数组时是 in place 操作，反复读写同一小块连续内存，数据有可能完全驻留在 L1 缓存甚至寄存器中，且没有分支预测之外的复杂流水线停顿，这比递归调用中不断跳转指令地址和跨栈帧访问对 CPU 流水线更友好。当然这是一个猜测，没有做实验验证。

从算法的角度上也可以进行分析，虽然从直觉上看，在 $O(N \log N)$ 的归并排序中混入 $O(N^2)$ 的冒泡排序似乎会降低效率，但实验结果已经证明并不是这样的。

时间复杂度 $O(\cdot)$ 描述的是算法随输入规模增长的趋势，而实际运行时间 $T(N)$ 可以更精确地建模为：

$$T(N) \approx C \cdot f(N) + K$$

其中 C 是常数因子，代表每次基本操作的实际硬件开销。

- **归并排序:** $T_{merge} \approx C_{high} \cdot N \log N$。
  这里的 C_{high} 非常大，因为它包含了函数调用栈帧的创建与销毁、昂贵的堆内存分配 (`malloc` / `free`) 以及辅助数组的数据拷贝。
- **冒泡排序:** $T_{bubble} \approx C_{low} \cdot N^2$。
  这里的 C_{low} 非常小，因为它是在寄存器和 L1 缓存中进行的简单比较和交换，没有额外的内存管理开销。

当 N 很小的时，尽管 $N^2 > N \log N$，但由于 $C_{high} \gg C_{low}$，使得不等式反转：

$$C_{low} \cdot N^2 < C_{high} \cdot N \log N$$

归并排序的执行过程构成了一棵二叉树。对于 N 个元素，完全递归树的节点总数约为 $2N - 1$，其中叶子节点的数量约为 N（占总节点的一半以上）。通过引入阈值 B，我们实际上剪掉了递归树最底部的 $\log_2 B$ 层，原本在这些层级发生的大量的 `merge` 调用被消除，取而代之的是在叶子节点上执行 N/B 次快速的线性遍历（冒泡排序）。

粗粒度化后的总时间复杂度大约为：

$$T(N) = \underbrace{O(N \log(\frac{N}{B}))}_{\text{上层归并开销}} + \underbrace{O(N \cdot B)}_{\text{底层基准排序开销}}$$

由于 B 是一个常数，第二项 $O(N \cdot B)$ 在渐进意义上退化为线性 $O(N)$。因此，整体算法的渐进时间复杂度依然保持在 $O(N \log N)$，但我们成功地极大地降低了隐藏在其中的常数因子。

同时我们也可以看出，如果 B 取得很大的话，和 N 呈一定比例，那么整体就会退化到 $O(N^2)$，导致更慢。

最优阈值

我还想知道最优 threshold 是多少，最简单的办法就是进行实验，我写了一个 python 脚本来批量测试并记录数据：

```python
import os
import re
import subprocess
import shutil
import sys

THRESHOLDS = [2, 4, 8, 16, 32, 64, 100, 128]

NUM_ELEMENTS = 400000
```

```python
NUM_TRIALS = 10

SOURCE_FILE = "sort_c.c"
BACKUP_FILE = "sort_c.c.bak"
EXECUTABLE = "./sort"

COMPILE_CMD = ["make", "sort"]
CLEAN_CMD = ["make", "clean"]

VALGRIND_CMD = [
    "valgrind",
    "--tool=cachegrind",
    "--cache-sim=yes",
    "--branch-sim=yes",
]

def backup_file():
    if os.path.exists(SOURCE_FILE):
        shutil.copy(SOURCE_FILE, BACKUP_FILE)

def restore_file():
    if os.path.exists(BACKUP_FILE):
        shutil.move(BACKUP_FILE, SOURCE_FILE)

def update_threshold(val):
    """使用正则修改源码中的阈值定义"""
    try:
        with open(SOURCE_FILE, 'r') as f:
            content = f.read()

        pattern = r"(#define\s+COARSEN_THRESHOLD\s+)(\d+)"
        if not re.search(pattern, content):
            print(f"[Error]")
            return False

        new_content = re.sub(pattern, f"\\g<1>{val}", content)

        with open(SOURCE_FILE, 'w') as f:
            f.write(new_content)
        return True
    except Exception as e:
        print(f"[Error] 修改文件失败: {e}")
        return False

def parse_valgrind_output(stdout_data, stderr_data):
    """解析 stdout 获取时间，解析 stderr 获取 Cachegrind 数据"""
    stats = {}

    time_match = re.search(r"sort_c\s+:\s+Elapsed execution time:\s+(\d+\.\d+)\s+sec",
stdout_data)
    stats['time'] = float(time_match.group(1)) if time_match else 0.0

    def get_metric(pattern, text):
        match = re.search(pattern, text)
        if match:
            return int(match.group(1).replace(',', ''))
        return 0

    stats['irefs'] = get_metric(r"I  refs:\s+([\d,]+)", stderr_data)
    stats['drefs'] = get_metric(r"D  refs:\s+([\d,]+)", stderr_data)
    stats['d1_miss'] = get_metric(r"D1\s+misses:\s+([\d,]+)", stderr_data)
    stats['branches'] = get_metric(r"Branches:\s+([\d,]+)", stderr_data)
```

```
    stats['mispred'] = get_metric(r"Mispredicts:\s+([\d,]+)", stderr_data)

    return stats

def main():
    print(f"--- Starting Advanced Benchmark (N={NUM_ELEMENTS}) ---")
    print(f"{'B':<4} | {'Time(s)':<8} | {'I refs':<14} | {'D refs':<14} | {'Branches':<14} |
{'Mispred':<10}")
    print("-" * 85)

    results = {}
    backup_file()

    try:
        for val in THRESHOLDS:
            if not update_threshold(val):
                break

            subprocess.run(CLEAN_CMD, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
            build = subprocess.run(COMPILE_CMD, stdout=subprocess.DEVNULL,
stderr=subprocess.PIPE)
            if build.returncode != 0:
                print(f"Build failed for threshold {val}")
                continue

            cmd = VALGRIND_CMD + [EXECUTABLE, str(NUM_ELEMENTS), str(NUM_TRIALS)]
            run = subprocess.run(cmd, capture_output=True, text=True)

            data = parse_valgrind_output(run.stdout, run.stderr)
            results[val] = data

            print(f"{val:<4} | {data['time']:.4f}   | {data['irefs']:<14,} | {data['drefs']:
<14,} | {data['branches']:<14,}| {data['mispred']:<10,}")

    except KeyboardInterrupt:
        print("\nInterrupted.")
    finally:
        restore_file()

    if results:
        best_time = min(results, key=lambda k: results[k]['time'])
        best_irefs = min(results, key=lambda k: results[k]['irefs'])

        print("-" * 85)
        print(f"Best Threshold (Time):   {best_time} ({results[best_time]['time']:.4f} sec)")
        print(f"Best Threshold (I refs): {best_irefs} ({results[best_irefs]['irefs']:,} refs)")

if __name__ == "__main__":
    main()
```

得到这样的实验表格：

```
--- Starting Advanced Benchmark (N=400000) ---
B    | Time(s)  | I refs         | D refs         | Branches       | Mispred
-------------------------------------------------------------------------------------
2    | 0.5259   | 4,643,787,481  | 1,556,803,604  | 431,781,375    | 20,769,228
4    | 0.3670   | 3,425,066,977  | 1,061,355,158  | 301,116,456    | 15,309,024
8    | 0.2962   | 2,805,673,942  | 817,248,953    | 246,744,579    | 16,680,793
16   | 0.2695   | 2,563,808,616  | 720,958,108    | 250,203,078    | 19,032,471
32   | 0.2820   | 2,652,566,704  | 744,320,384    | 319,187,358    | 22,363,141
64   | 0.3545   | 3,225,481,179  | 930,821,490    | 500,381,546    | 32,450,513
```

```
100  | 0.5203   | 4,593,237,593  | 1,376,164,746  | 881,391,378   | 58,995,373
128  | 0.5190   | 4,593,237,608  | 1,376,164,751  | 881,391,383   | 58,995,377
-------------------------------------------------------------------------------------
Best Threshold (Time):   16 (0.2695 sec)
Best Threshold (I refs): 16 (2,563,808,616 refs)
```

可以看到 8,16,32 的数据都差不多，时间最快的是 16（刚才真的是随便选的）。这说明 16 刚好是归并开销与冒泡开销的一个平衡点。除此之外，$B = 16$ 可能还有一个硬件优势，当 $B = 16$ 时，子数组的总大小为 $16 \times 4 = 64$ 字节，而现代 CPU Cache Line 大小通常正是 64 字节，减少了很多搬运的开销。

理论最优阈值推导

其实在开始实验前，我本来想沿着算法分析，从理论上算一下最优的 threshold，虽然结果说明这样是不行的，但我觉得这个推导很有意思，故在此贴出。

设总运行时间 $T(B)$ 由归并开销和基准排序开销组成：

$$T(B) \approx \underbrace{C_{merge} \cdot N \cdot (\log_2 N - \log_2 B)}_{\text{剩余的归并层工作}} + \underbrace{C_{base} \cdot N \cdot B}_{\text{底层冒泡排序工作}}$$

其中：

- C_{merge}: 归并一层所需的平均单元素开销（包含 `malloc`、`free`、数据移动）。
- C_{base}: 冒泡排序的常数因子（冒泡排序是 $O(B^2)$，但在 N/B 个块上运行，总复杂度为 $\frac{N}{B} \cdot B^2 = N \cdot B$）。

为了找到使时间 $T(B)$ 最小的 B，我们对 B 求导并令其为 0：

$$\frac{dT}{dB} = C_{merge} \cdot N \cdot (-\frac{1}{B \ln 2}) + C_{base} \cdot N = 0$$

解得最优阈值 B_{opt}：

$$B_{opt} = \frac{C_{merge}}{C_{base} \cdot \ln 2}$$

从公式可以看出，B_{opt} 与数据总量 N 完全无关，它仅取决于归并操作开销与基准排序开销的比值。

这个结果其实没啥用，但是我们通过实验找到了最优的 threshold，可以通过这个值反推出 C_{merge} 和 C_{base} 的比值，也算是一个小应用。

Write-up 6

内存访问优化

```c
#include "./util.h"

#define COARSEN_THRESHOLD 16

static void merge_m(data_t* A, int p, int q, int r);
static void copy_m(data_t* source, data_t* dest, int n);
static void bubble_sort_m(data_t* A, int p, int r);

void sort_m(data_t* A, int p, int r) {
  assert(A);

  int len = r - p + 1;

  if (len <= COARSEN_THRESHOLD) {
      bubble_sort_m(A, p, r);
  }
  else if (p < r) {
    int q = (p + r) / 2;
    sort_m(A, p, q);
```

```
    sort_m(A, q + 1, r);
    merge_m(A, p, q, r);
  }
}

static void bubble_sort_m(data_t* A, int p, int r) {
    int len = r - p + 1;
    for (int i = 0; i < len - 1; i++) {
        for (int j = p; j < r - i; j++) {
            if (A[j] > A[j+1]) {
                data_t temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}

static void merge_m(data_t* A, int p, int q, int r) {
  assert(A);
  assert(p <= q);
  assert((q + 1) <= r);

  int n1 = q - p + 1;

  data_t* left = 0;

  // 仅分配 left 的内存
  mem_alloc(&left, n1);

  if (left == NULL) {
    return;
  }

  // 仅将左半部分 (A[p...q]) 拷贝到临时空间 left
  copy_m(A + p, left, n1);

  data_t *l_ptr = left;              // 指向临时空间 left 的开头
  data_t *l_end = left + n1;         // left 的结束边界

  data_t *r_ptr = A + q + 1;         // 右半部分直接读取原数组 A
  data_t *r_end = A + r + 1;         // right (在 A 中) 的结束边界

  data_t *dest_ptr = A + p;          // 写入位置，从 A[p] 开始

  while (l_ptr < l_end && r_ptr < r_end) {
    if (*l_ptr <= *r_ptr) {
      *dest_ptr++ = *l_ptr++;
    } else {
      *dest_ptr++ = *r_ptr++;
    }
  }

  // 只需要处理 left 剩余的情况。
  // 如果 right (在 A 中) 有剩余，它们本来就在数组的后半部分，无需移动
  while (l_ptr < l_end) {
    *dest_ptr++ = *l_ptr++;
  }

  mem_free(&left);
}
```

```
static void copy_m(data_t* source, data_t* dest, int n) {
  assert(dest);
  assert(source);

  data_t* source_end = source + n;
  while (source < source_end) {
    *dest++ = *source++;
  }
}
```

性能分析

根据作业要求，我们先分析一下 `sort_m.c` 是否有性能提升？

运行：

```
make clean
make sort
valgrind --tool=cachegrind --cache-sim=yes --branch-sim=yes ./sort 400000 10
```

数据：

```
==39869== Cachegrind, a high-precision tracing profiler
==39869== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==39869== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==39869== Command: ./sort 400000 10
==39869==
--39869-- warning: L3 cache found, using its data for the LL simulation.
--39869-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--39869-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_m          : Elapsed execution time: 0.279528 sec
Generating inverted array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_m          : Elapsed execution time: 0.474017 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==39869==
==39869== I refs:        2,421,063,381
==39869== I1  misses:            1,735
==39869== LLi misses:            1,643
==39869== I1  miss rate:          0.00%
==39869== LLi miss rate:          0.00%
==39869==
==39869== D refs:          558,768,695  (328,504,949 rd   + 230,263,746 wr)
==39869== D1  misses:       12,123,025  (  6,531,877 rd   +   5,591,148 wr)
==39869== LLd misses:           76,722  (      1,287 rd   +      75,435 wr)
==39869== D1  miss rate:           2.2% (        2.0%     +         2.4%  )
==39869== LLd miss rate:           0.0% (        0.0%     +         0.0%  )
==39869==
==39869== LL refs:          12,124,760  (  6,533,612 rd   +   5,591,148 wr)
```

```
==39869== LL misses:            78,365  (      2,930 rd   +      75,435 wr)
==39869== LL miss rate:            0.0% (        0.0%     +         0.0%  )
==39869==
==39869== Branches:        347,150,426  (341,838,964 cond +   5,311,462 ind)
==39869== Mispredicts:      19,305,554  ( 19,305,255 cond +         299 ind)
==39869== Mispred rate:            5.6% (        5.6%     +         0.0%   )
```

和 `sort_c.c` 做一下对比：

性能指标 (Metric)	sort_c (Coarsening)	sort_m (Memory Opt)	变化趋势 (Delta)
随机数组耗时 (Time - Random)	0.267813 sec	0.279528 sec	+4.3%
倒序数组耗时 (Time - Inverted)	0.522963 sec	0.474017 sec	-9.3%
总指令数 (I refs)	2,563,812,419	2,421,063,381	-5%
数据引用/内存访问 (D refs)	**720,959,502**	**558,768,695**	-22.5%
L1 数据缓存未命中 (D1 misses)	15,943,300	12,123,025	-24%
LL (L3) 数据缓存未命中 (LLd misses)	101,777	76,722	-24.6%
分支指令数 (Branches)	**250,203,844**	**347,150,426**	+38.7%
分支预测错误 (Mispredicts)	19,032,537	19,305,554	略微增加
分支预测错误率 (Mispred rate)	7.6%	5.6%	-1.9%

我们可以看到运行时间没什么区别，但是内存访问和未命中数确实有显著减少。这说明内存优化确实是有效的，通过移除 `right` 临时数组的分配，并直接利用输入数组 `A` 的空间来存储右半部分数据，可以消除了约一半的内存分配开销和一半的内存拷贝操作 (`copy_c`)，极大地减少了内存流量。

分支指令 (`Branches`) 却急剧增加了 38.7%，我猜测是因为移除了哨兵值 (`UINT_MAX`) 造成的。在 `sort_c` 中，哨兵的使用使得合并循环可以用更少的条件检查运行。而在 `sort_m` 中，为了安全地使用输入数组 `A` 而不发生越界或错误覆盖，我们必须在 `while` 循环的每次迭代中对两个指针实现显式的边界检查 (`l_ptr < l_end && r_ptr < r_end`)，这导致了更多的逻辑判断开销。

对于随机数据，CPU 处理大幅增加的分支指令所带来的开销，抵消并略微超过了减少缓存未命中带来的收益，所以整体甚至会更慢一点，但是也不算很多。

对于倒序情况，通常涉及排序过程中最大量的数据移动，内存带宽会十分影响性能。`sort_m` 22.5% 的内存访问减少的收益超过了分支开销的负面影响，从而带来了明显的速度提升。

编译器能否自动进行这种优化?

从直觉来看，很显然是不能的，编译器还没有这么“智能”，而且这应该属于篡改代码语义了，编译器是不能也万万不可这么做的。

更学术一点，这种优化需要对算法的逻辑和内存管理策略进行根本性的改变，而不仅仅是局部的代码微调。而且将基于哨兵的循环转换为基于边界检查的循环会显著改变程序的控制流。编译器通常必须严格保留原始的控制流逻辑以确保程序的正确性，它无法推断出在此特定上下文中移除哨兵是安全或预期的优化行为，这也是为了程序安全它们必须做到的事情。

Write-up 7

代码

这个问题要求我们紧接着 Write-up 6 做进一步优化，在merge函数被调用之前就申请一大块内存，然后在这块内存上进行后续的操作，直至完成后再释放，这样可以进一步减少时间开销。

```c
#include "./util.h"

#define COARSEN_THRESHOLD 16
```

```c
static void sort_f_helper(data_t* A, int p, int r, data_t* temp);
static void merge_f(data_t* A, int p, int q, int r, data_t* temp);
static void copy_f(data_t* source, data_t* dest, int n);
static void bubble_sort_f(data_t* A, int p, int r);

void sort_f(data_t* A, int p, int r) {
  assert(A);

  int len = r - p + 1;
  if (len <= 0) return;

  data_t* temp = 0;
  mem_alloc(&temp, len);

  if (temp == NULL) {
    return;
  }

  sort_f_helper(A, p, r, temp);

  mem_free(&temp);
}

static void sort_f_helper(data_t* A, int p, int r, data_t* temp) {
  int len = r - p + 1;

  if (len <= COARSEN_THRESHOLD) {
      bubble_sort_f(A, p, r);
  }
  else if (p < r) {
    int q = (p + r) / 2;
    sort_f_helper(A, p, q, temp);
    sort_f_helper(A, q + 1, r, temp);
    merge_f(A, p, q, r, temp);
  }
}

static void merge_f(data_t* A, int p, int q, int r, data_t* temp) {

  int n1 = q - p + 1;

  data_t* left = temp;

  copy_f(A + p, left, n1);

  data_t *l_ptr = left;
  data_t *l_end = left + n1;

  data_t *r_ptr = A + q + 1;
  data_t *r_end = A + r + 1;

  data_t *dest_ptr = A + p;

  while (l_ptr < l_end && r_ptr < r_end) {
    if (*l_ptr <= *r_ptr) {
      *dest_ptr++ = *l_ptr++;
    } else {
      *dest_ptr++ = *r_ptr++;
    }
  }
```

```
  while (l_ptr < l_end) {
    *dest_ptr++ = *l_ptr++;
  }
}

static void bubble_sort_f(data_t* A, int p, int r) {
    int len = r - p + 1;
    for (int i = 0; i < len - 1; i++) {
        for (int j = p; j < r - i; j++) {
            if (A[j] > A[j+1]) {
                data_t temp_val = A[j];
                A[j] = A[j+1];
                A[j+1] = temp_val;
            }
        }
    }
}

static void copy_f(data_t* source, data_t* dest, int n) {
  data_t* source_end = source + n;
  while (source < source_end) {
    *dest++ = *source++;
  }
}
```

然后进行性能测试

```
==77488== Cachegrind, a high-precision tracing profiler
==77488== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==77488== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==77488== Command: ./sort 400000 10
==77488==
--77488-- warning: L3 cache found, using its data for the LL simulation.
--77488-- warning: specified LL cache: line_size 64  assoc 12  total_size 18,874,368
--77488-- warning: simulated LL cache: line_size 64  assoc 18  total_size 18,874,368

Running test #0...
Generating random array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_f          : Elapsed execution time: 0.256708 sec
Generating inverted array of 400000 elements
Arrays are sorted: yes
 --> test_correctness at line 217: PASS
sort_f          : Elapsed execution time: 0.429010 sec

Running test #1...
 --> test_zero_element at line 245: PASS

Running test #2...
 --> test_one_element at line 266: PASS
Done testing.
==77488==
==77488== I refs:        2,313,226,341
==77488== I1  misses:            1,739
==77488== LLi misses:            1,643
==77488== I1  miss rate:          0.00%
==77488== LLi miss rate:          0.00%
==77488==
==77488== D refs:          508,216,031  (298,062,524 rd   + 210,153,507 wr)
==77488== D1  misses:       12,058,890  (  6,501,927 rd   +   5,556,963 wr)
```

```
==77488== LLd misses:           76,697  (      1,290 rd   +      75,407 wr)
==77488== D1  miss rate:           2.4% (        2.2%     +         2.6%  )
==77488== LLd miss rate:           0.0% (        0.0%     +         0.0%  )
==77488==
==77488== LL refs:          12,060,629  (  6,503,666 rd   +   5,556,963 wr)
==77488== LL misses:            78,340  (      2,933 rd   +      75,407 wr)
==77488== LL miss rate:            0.0% (        0.0%     +         0.0%  )
==77488==
==77488== Branches:        332,024,225  (328,023,399 cond +   4,000,826 ind)
==77488== Mispredicts:      18,225,229  ( 18,224,930 cond +         299 ind)
==77488== Mispred rate:            5.5% (        5.6%     +         0.0%   )
```

和 write-up 6 对比可以发现基本区别不大，运行时间略快，总指令数少了一些。不过还是有提升，这种提升估计还是来源于消除了内存分配器的一些系统开销。

不过由于归并排序的核心算法逻辑没有改变，只是改变了临时内存的生命周期，所以可以看到 Cache Miss 和分支预测数据基本没啥变化。

总结和感想

由于还要写其他作业，选做的几题没时间做更多实验和探究了，颇为遗憾。我觉得本次实验最有意思的点是以小见大，以一个我们大一就学过的归并排序算法为基础，一步步改善，并教我们如何结合 cachegrind 工具来做性能分析，看这些优化是否对性能产生了真正的影响，以及为什么？在这过程中我收获颇丰，也结合起了很多以前在 csapp 学到的知识，感觉在这个 ai 主流时代这些知识终于有了用武之地，也是比较感慨。虽然现在大家都在卷 ai，但是底层的系统优化还是非常重要的，一个小小的改进就有可能对程序性能产生巨大影响，这对高度依赖计算的 ai 体系来说其实至关重要，也希望遗以后大家能够多关注关注做系统的，不要忘了来时路。感想写的比较随意和流水账，助教老师见笑了。