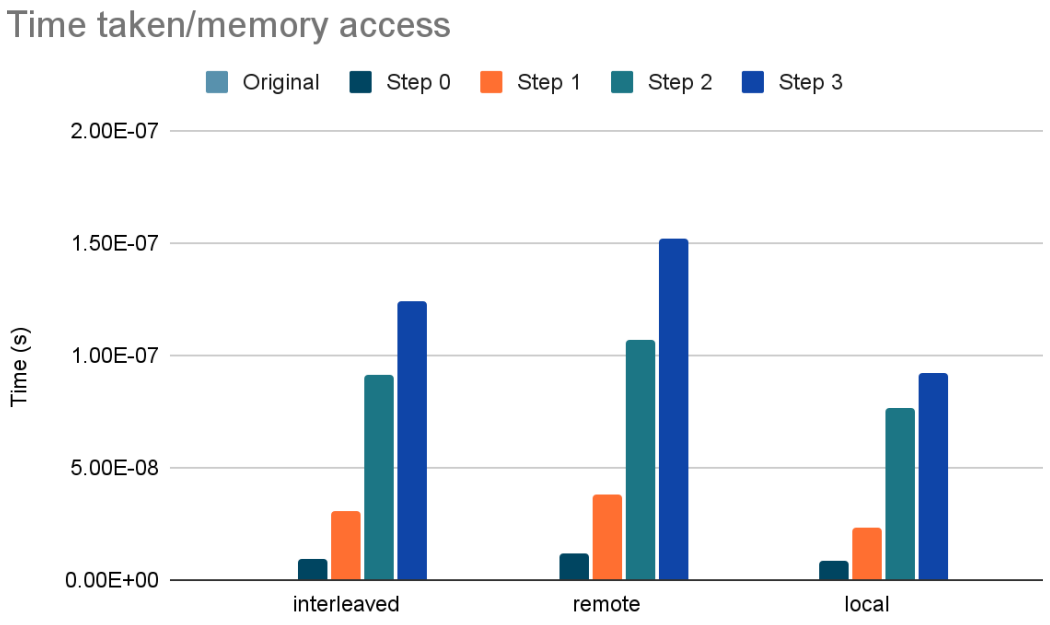
Assignment 3

Wenkai Li, Zehao Chen

1. We present the execution time when running with different NUMA policies in the table below. The command used are as follow:
Interleaved: `numactl --interleave=all ./numa`
Remote: `numactl --cpunodebind=0 --membind=1 ./numa`
Local: `numactl --localalloc ./numa`

We can see that the execution time with three policies follows that remote > interleaved > local.
Local policy yields the shortest time per memory access, because memory is always allocated on the current node with the lowest latency.
Remote policy gives the longest time per memory access, because memory is always allocated on the node in the other socket, thus introducing the highest latency when fetching data from memory.
The performance of interleaved policy sits between the other two policies, because some memories are allocated on the local node, while others are allocated on the remote node, giving median time per memory access.

	Original	Step 0	Step 1	Step 2	Step 3
interleaved	3.08e-10	9.68e-09	3.05e-08	9.12e-08	1.24e-07
remote	3.24e-10	1.19e-08	3.78e-08	1.07e-07	1.52e-07
local	3.01e-10	8.37e-09	2.37e-08	7.63e-08	9.20e-08



2. We deoptimized the numa program in four steps. Results are presented in the table above.

Step 0: We remove the benefit of caching.
We set each array element to be 64 (size of a cache block because each uint8_t takes one byte) and let next _addr return arr[i]. In this way, we always read the 64th element from the current one which will be in a different cache block, therefore we will not benefit from a previous cache block that has been fetched.
This de-optimization brings access time close to 1e-08 level, about 30 times slower than the original program.

Step 1: We remove the benefit of cache line prefetching.
We set each array element to be 128. If the prefetcher only prefetches one next cache line, we will skip access to that cache block as well with this de-optimization.
This roughly triples the access time observed in step 0.

Step 2: We break linear patterns in memory access and remove the benefit of stride prefetching.
We set each array element to be 128 plus a random number in range(0, 127). In this way, the next element we read could be in either the 3rd or 4th cache block, without a fixed pattern.
This roughly triples the access time again compared to step 1.

Step 3: We add extra calculation of the next memory address.
We let next_addr return arr[i] * 2, because we want to examine the effect of forced serial calculation of next address.
This gives roughly another 50% increase in the memory access time.

We did not actually anticipate this much de-optimization compared to step 2. 50% increase seems a little too much for a simple multiplication, but this is the only meaningful difference we can see from examining assembly output.
Conceptually we believe that in step 2 (or even in step 0, where we let next_addr return arr[i] instead of an integer literal, which on its own also gave significant slowdown), the random array elements would already guarantee that the address of the next access can only be calculated after the previous access completes.

3. In the assembly output of the original program, we have:

```
thread1Func:
      movb  $1, t1block(%rip)
      movl  $1, X(%rip)           // X = 1
      movl  Y(%rip), %eax         // load Y into register
      movl  %eax, r1(%rip)        // r1 = Y
      movb  $1, t1fin(%rip)
thread2Func:
      movb  $1, t2block(%rip)
      movl  $1, Y(%rip)           // Y = 1
      movl  X(%rip), %eax         // load X into register
```

```
movl %eax, r2(%rip)          // r2 = X
movb $1, t2fin(%rip)
```

The instructions marked red could be reordered within each thread, because they do not depend on each other. The real execution order could be as follows where we would detect reordering:

```
movl Y(%rip), %eax
movl X(%rip), %eax
movl %eax, r1(%rip)
movl %eax, r2(%rip)
movl $1, X(%rip)
movl $1, Y(%rip)
```

To disable memory reorderings, we changed

```
asm volatile("" ::: "memory");
```

to

```
asm volatile("MFENCE" ::: "memory");
```

The first instruction is only to prevent compiler reordering, while the latter can prevent both compiler and CPU reordering.

The assembly output now has a MFENCE between the instructions, and prevents reordering while executing (irrelevant instructions like #APP #NO_APP removed):

```
thread1Func:
      movb $1, t1block(%rip)
      movl $1, X(%rip)           // X = 1
      MFENCE
      movl Y(%rip), %eax         // load Y into register
      movl %eax, r1(%rip)        // r1 = Y
      movb $1, t1fin(%rip)
thread2Func:
      movb $1, t2block(%rip)
      movl $1, Y(%rip)           // Y = 1
      MFENCE
      movl X(%rip), %eax         // load X into register
      movl %eax, r2(%rip)        // r2 = X
      movb $1, t2fin(%rip)
```

After the change, the processor would block when instruction reaches MFENCE. It will wait for the store buffer to drain and only then move on to the load instruction.

4. We present the percentage of memory reorderings in the table below. The command used are as follows:

```
numactl --physcpubind=0,1,2 ./order
numactl --physcpubind=0,1,18 ./order
```

	Before	After
Same socket	98%	0%
Different socket	42%	0%

In the original program, we observe memory reordering in 98% of the iterations when running two threads on the same socket, but only in 42% of the iterations when running two threads on different sockets. After adding the memory fence, we observe no reordering in both cases.
We observe much lower reorderings when running on different sockets, because loading data from a different socket has higher latency. The store operation may actually complete before the load. Therefore, we got the result that fewer reorderings had taken place.