OpenMP: A Programmer's Perspective

Advanced Compiler / Term Presentation

2023 / 06 / 08

박찬우

OpenMP

- An API for shared-memory parallelism in C, C++ and Fortran programs
  - A set of compiler directives, library routines, and environment variables for parallel application programmers
- Fork-Join Parallelism
  - The master thread spawns a team of threads.
  - When the team of threads complete the work in the parallel section, they terminate synchronously, leaving only the master thread.
- Compiler generates thread program and synchronization
  - Programmer should “explicitly” define code sections to be parallelized.
- OpenMP is an *API*
  - Implementation detail might vary.

Reference: Lecture Note(2023 봄학기, 확장형 고성능 컴퓨팅)

Shared-Memory Parallelism

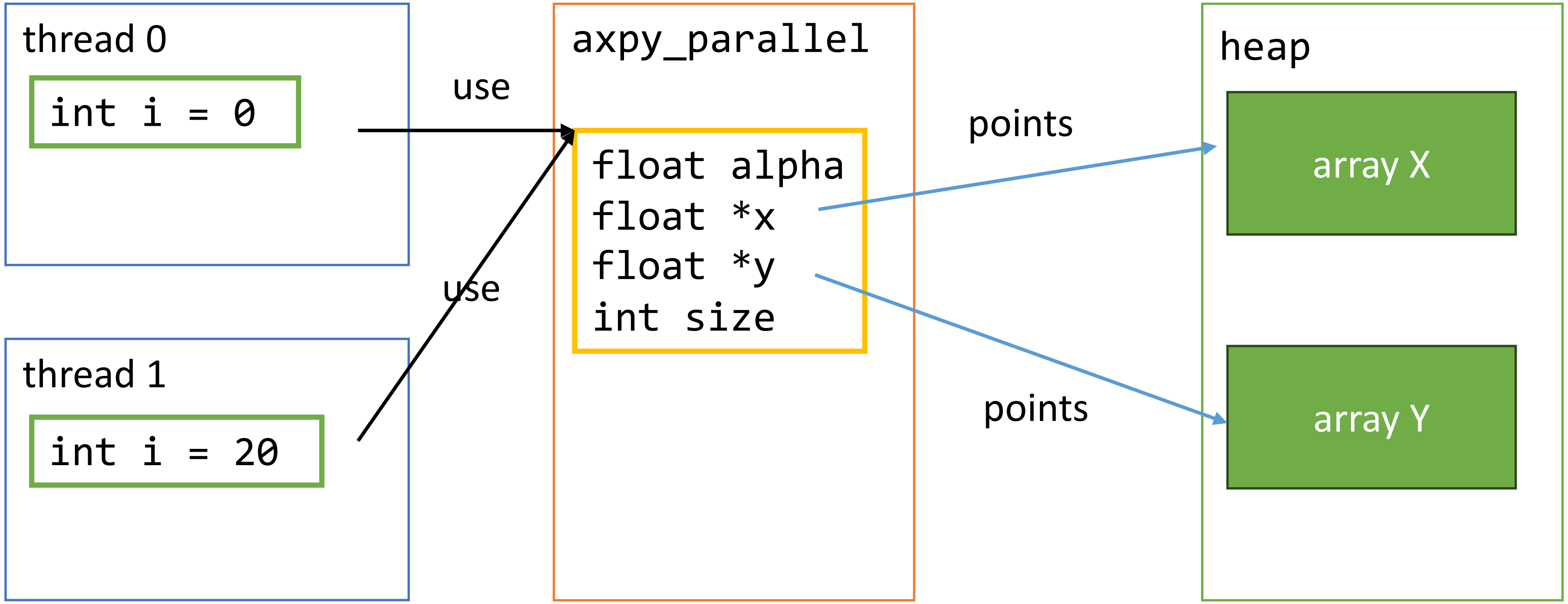
- An API for shared-memory parallelism in C, C++ and Fortran programs
  - A set of compiler directives, library routines, and environment variables for parallel application programmers
- In an OpenMP program, threads have shared variables and private variables

```
void axpy_parallel(float alpha, float* x, float* y, int size){
#pragma omp parallel for
      for (int i = 0; i < size; i++) {
            y[i] = y[i] + alpha * x[i];
      }
}
```

Shared-Memory Parallelism(Cont’d)

- In an OpenMP program, threads have shared variables and private variables

```
void axpy_parallel(float alpha, float* x, float* y, int size){
#pragma omp parallel for
      for (int i = 0; i < size; i++) {y[i] = y[i] + alpha * x[i];}}
```



OpenMP Compiler Directive

- An API for shared-memory parallelism in C, C++ and Fortran programs
  - A set of compiler directives, library routines, and environment variables for parallel application programmers

```
void axpy_parallel(float alpha, float* x, float* y, int size){
#pragma omp parallel for
       for (int i = 0; i < size; i++) {
             y[i] = y[i] + alpha * x[i];
       }
}
```

OpenMP Compiler Directive(Cont’d)

- An API for shared-memory parallelism in C, C++ and Fortran programs
  - A set of compiler directives, library routines, and environment variables for parallel application programmers
  - reduction clause provides a private copy of a shared variable to each thread, then reduces the copy to the shared variable with a synchronization mechanism.

```
float sdot_parallel(float* x, float* y, int size){
      float sum = 0.0;
#pragma omp parallel for reduction(+: sum)
      for (int i = 0; i < size; i++) {
            sum += x[i] * y[i]; // has loop-carried true dependence
      }
}
```

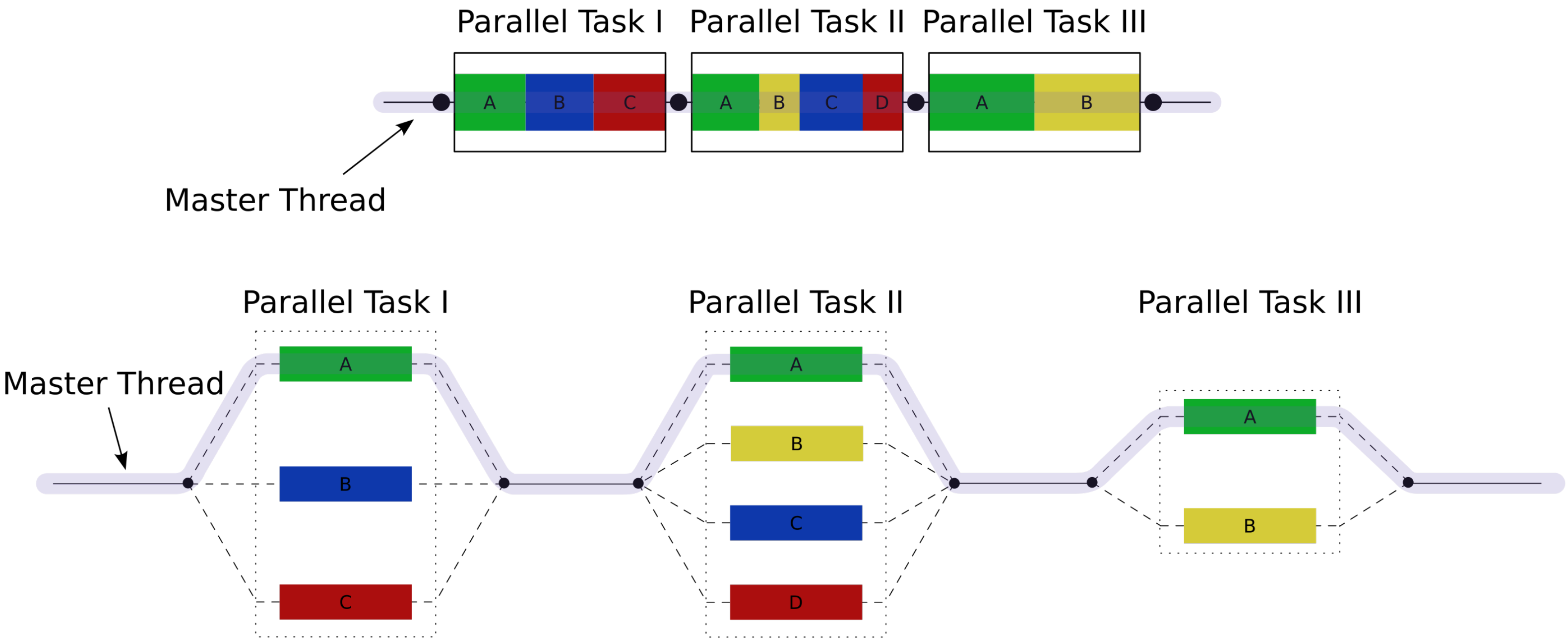
OpenMP

- Fork-Join Parallelism
  - The master thread spawns a team of threads.
  - When the team of threads complete the work in the parallel section, they terminate synchronously, leaving only the master thread.
- Compiler generates thread program and synchronization
  - Programmer should “explicitly” define code sections to be parallelized.
- OpenMP is an *API*
  - Implementation detail might vary.

Reference: Lecture Note(2023 봄학기, 확장형 고성능 컴퓨팅)

Fork-Join Parallelism

- The master thread spawns a team of threads.
- When the team of threads complete the work in the parallel section, they terminate synchronously, leaving only the master thread.



Reference: Lecture Note(2023 봄학기, 확장형 고성능 컴퓨팅), Image from Wikipedia(fork-join model)

OpenMP

- The compiler generates the thread program and the synchronization.
  - Programmers should "explicitly" define parallel region.
- OpenMP is an *API*
  - Implementation detail might vary.

    (That's all for the paper, OpenMP: An Industry Standard API for Shared Memory Programming)

Reference: Lecture Note(2023 봄학기, 확장형 고성능 컴퓨팅)

GCC Implementation for OpenMP Support

- The compiler generates the thread program and the synchronization.

```
void axpy_parallel(float alpha,
float* x, float* y, int size){
#pragma omp parallel for
for (int i = 0; i < size; i++) {
  y[i] = y[i] + alpha * x[i];
  }
}
```

```
axpy_parallel:
sub rsp, 40
xor ecx, ecx
mov DWORD PTR [rsp+20], edx
xor edx, edx
mov QWORD PTR [rsp+8], rsi
mov rsi, rsp
mov QWORD PTR [rsp], rdi
mov edi, OFFSET FLAT:axpy_parallel._omp_fn.0
movss DWORD PTR [rsp+16], xmm0
call GOMP_parallel
add rsp, 40
ret
```

```
axpy_parallel._omp_fn: An outline function of the parallel region

      GOMP_parallel(axpy_parallel._omp_fn, stack pointer, 0, 0)
// gcc/libomp/parallel.c: defines the following function
void GOMP_parallel (void (*fn) (void *), void *data, unsigned num_threads,
unsigned int flags)
```

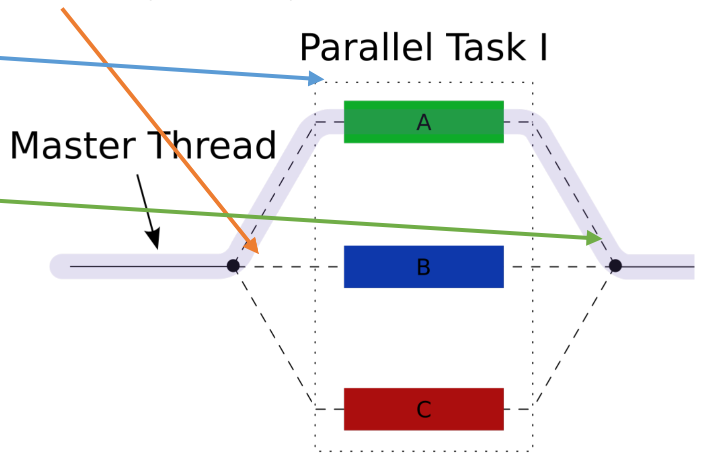
Compile option: -O2 –fopenmp x86-64 gcc 13.1 // https://godbolt.org/

GCC Implementation for OpenMP Support (Cont’d)

```
void axpy_parallel(float alpha, float* x, float* y, int size){
#pragma omp parallel for
  for (int i = 0; i < size; i++){y[i] = y[i] + alpha * x[i];}}
```

```
GOMP_parallel(axpy_parallel._omp_fn, stack pointer, 0, 0)
```

```
// gcc/libomp/parallel.c: defines the following function
void GOMP_parallel (void (*fn) (void *), void *data, unsigned num_threads, unsigned int flags){
 num_threads = gomp_resolve_num_threads (num_threads, 0);
 // master thread starts the thread team
 gomp_team_start (fn, data, num_threads, flags, gomp_new_team (num_threads), NULL);
  // master thread executes the parallel region
 fn (data);
// master thread joins the thread team
 ialias_call (GOMP_parallel_end) ();
}
```



Compile option: -O2 –fopenmp x86-64 gcc 13.1 // https://godbolt.org/

Key Takeaway

- OpenMP is an API for shared-memory parallelism in C, C++, and Fortran programs
  - A set of compiler directives, library routines, and environment variables for parallel application programmers
- The compiler generates the thread program and the synchronization.
  - The master thread creates a team of threads, then joins
  - Programmers should “explicitly” define the parallel region.
- This API is (usually) implemented with multiple functions.
  - GCC / Clang
  - They outline internal code block

Compiling a Program with OpenMP

- `clang -o main main.c -O2 -fopenmp=libomp`
- `gcc -o main main.c -O2 -fopenmp`

Motivating Example

- Let’s parallelize code blocks below.

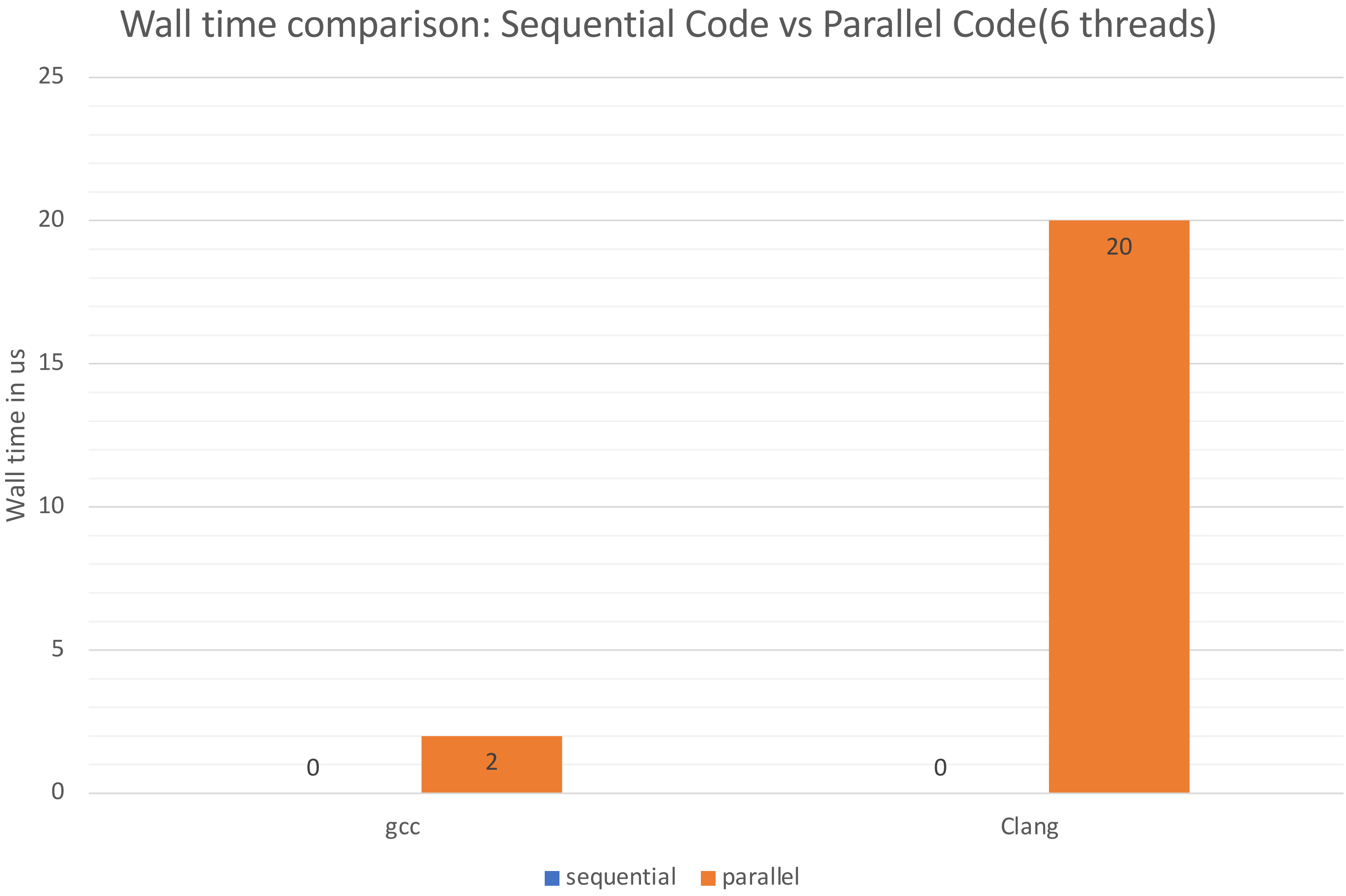
```
void init(float a, float b, float* vec, int size){
   for (int i = 0; i < size; i++) vec[0] = compute(a, b);
}
void parallel_init(float a, float b, float* vec, int size){
#pragma omp parallel for
   for (int i = 0; i < size; i++) vec[0] = compute(a, b);
}
```

[[Note]] Yes, the example is far from a real world application.

We use this code to focus on ”what a compiler can do”.

`compute(a,b)` is a macro (Not a function)

Unexpected Result



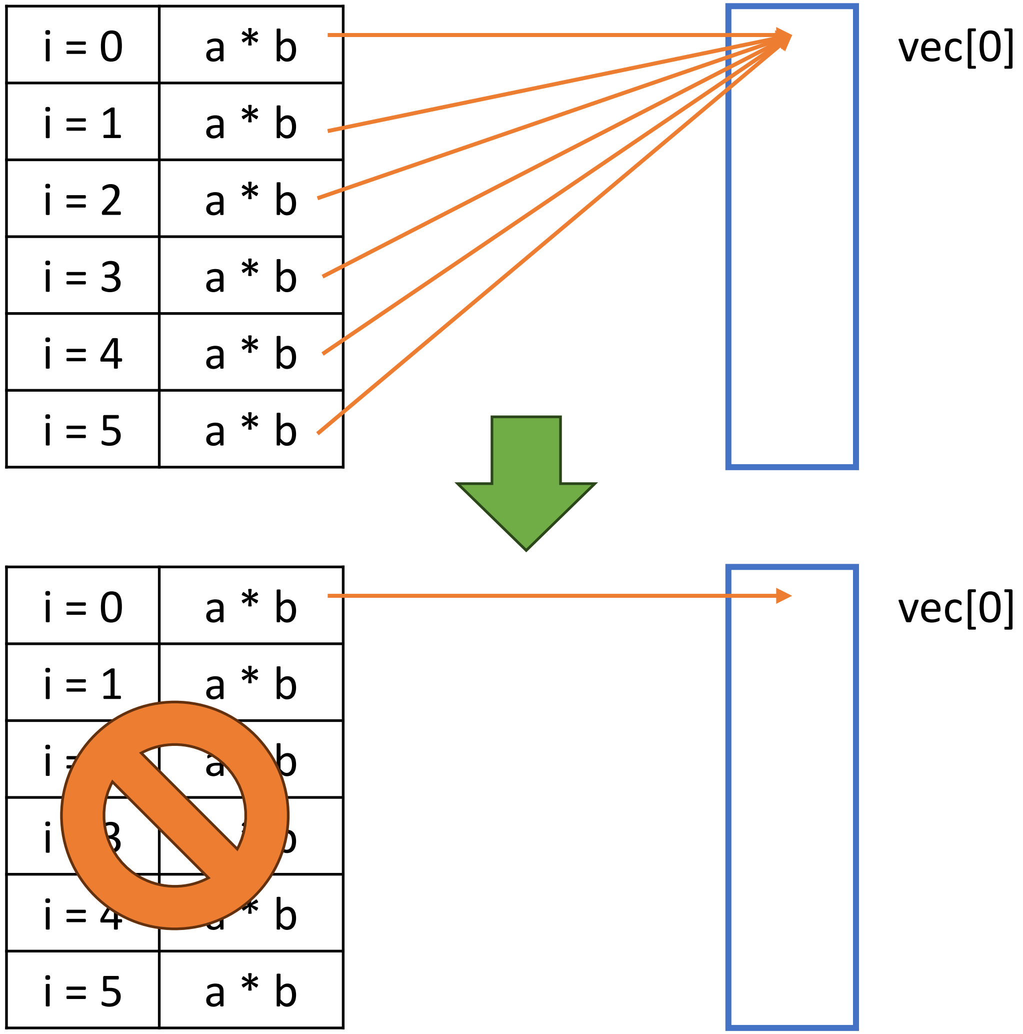
Caveat 1: Dead Code Elimination

```
#pragma omp parallel for
for (int i = 0; i < size; i++)
        vec[0] = compute(a, b);
```

Ideal: remove the for statement

In sequential codes, the for statement

is eliminated.



i = 0	a * b
i = 1	a * b
i = 2	a * b
i = 3	a * b
i = 4	a * b
i = 5	a * b

vec[0]

i = 0	a * b
i = 1	a * b
i = 2	a * b
i = 3	a * b
i = 4	a * b
i = 5	a * b

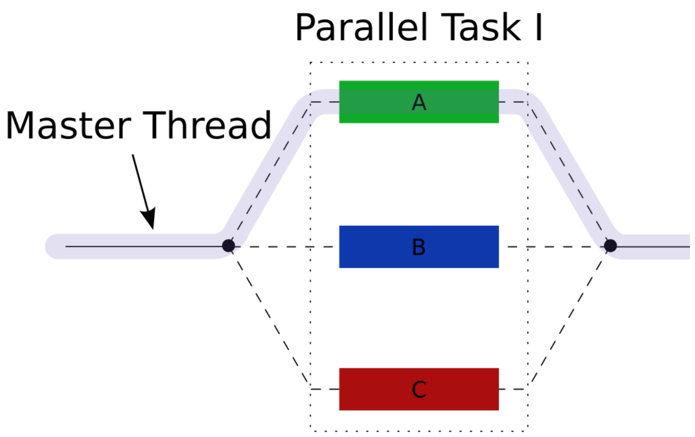
vec[0]

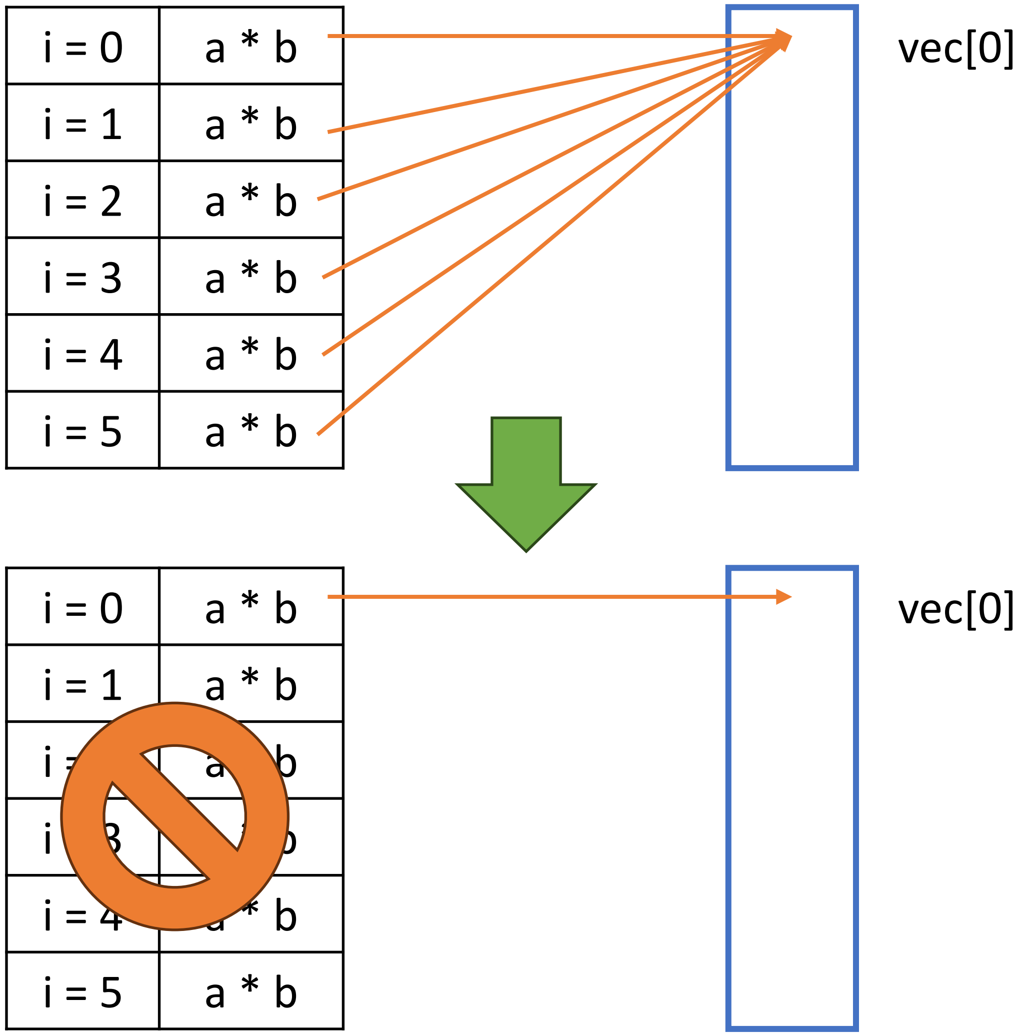
Caveat 1: Dead Code Elimination (Cont'd)

```
#pragma omp parallel for
for (int i = 0; i < size; i++)
        vec[0] = compute(a, b);
```

Why can't we eliminate for statement?

- The OpenMP runtime spawns a team of threads.
- Behavior changes





i = 0	a * b
i = 1	a * b
i = 2	a * b
i = 3	a * b
i = 4	a * b
i = 5	a * b

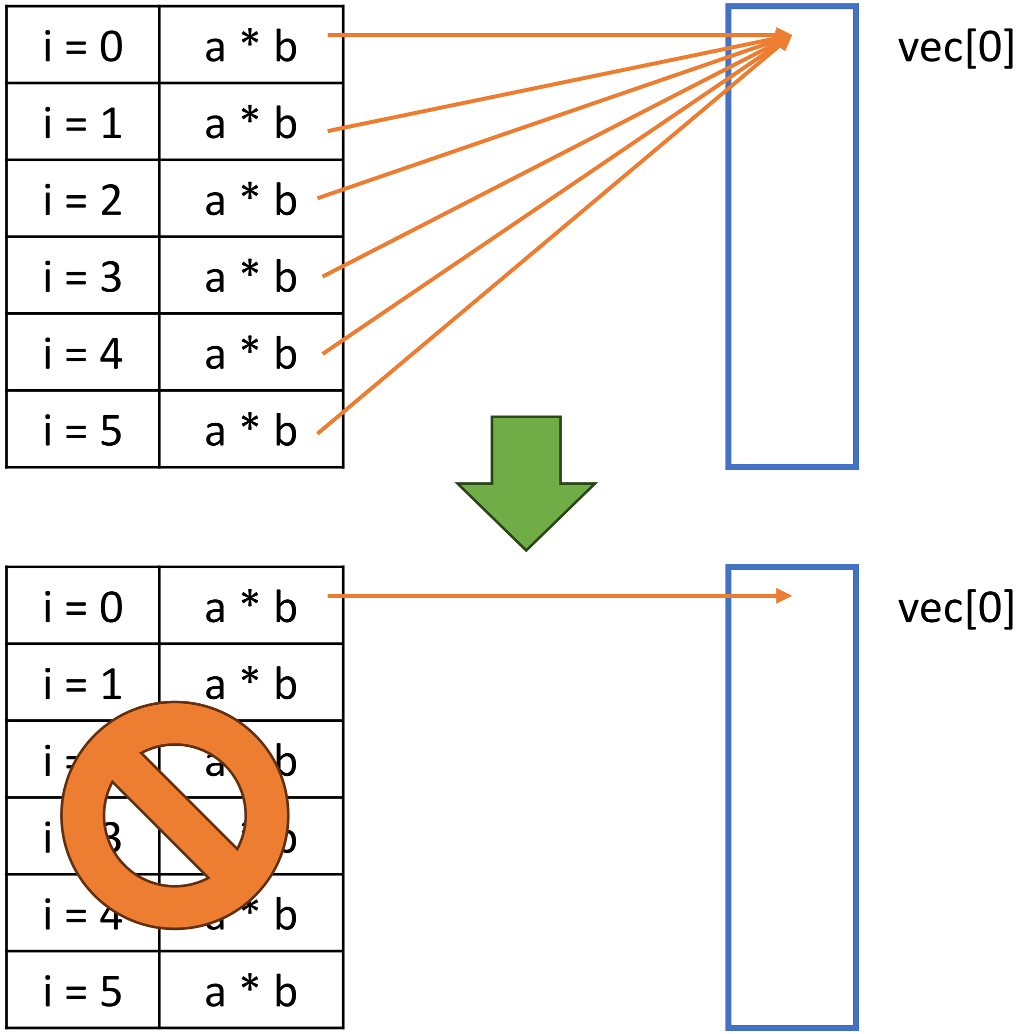
i = 0	a * b
i = 1	a * b
i	a b
i 3	b
i = 4	* b
i = 5	a * b

Caveat 1: Dead Code Elimination (Cont'd)

```
for (int i = 0; i < size; i++)
        vec[0] = compute(a, b);
```

Takeaway

- (1) Manually inspect code blocks if the block is necessary.
- (2) Do not parallelize what you don't need to.



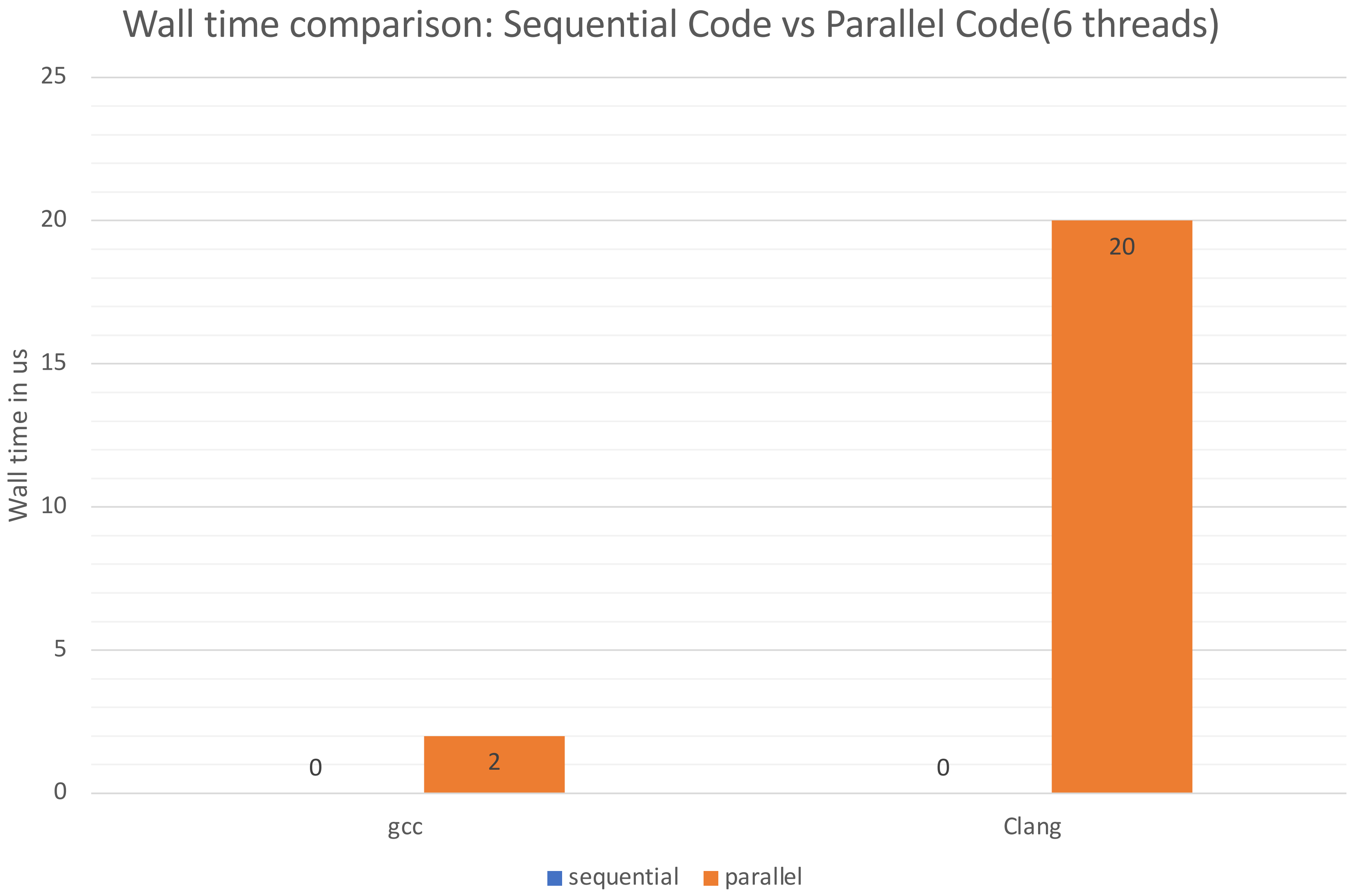
i = 0	a * b
i = 1	a * b
i = 2	a * b
i = 3	a * b
i = 4	a * b
i = 5	a * b

vec[0]

i = 0	a * b
i = 1	a * b
i	a b
i 3	b
i = 4	* b
i = 5	a * b

vec[0]

Observation: Binary Generated by Clang is Runs Slower!



Caveat 2: Variable Privatization(Cont’d)

```
#pragma omp parallel for
for (int i = 0; i < size; i++)
  vec[0] = a * b;
```

We know that `a * b` is loop invariant.

However, the code

- loads **a**

  `(movss xmm0, dword ptr [rbx])`

- loads **b** and multiplies,

  `(movss xmm0, dword ptr [rbx])`

- then stores the result.

  `(movss dword ptr [rax], xmm0)`

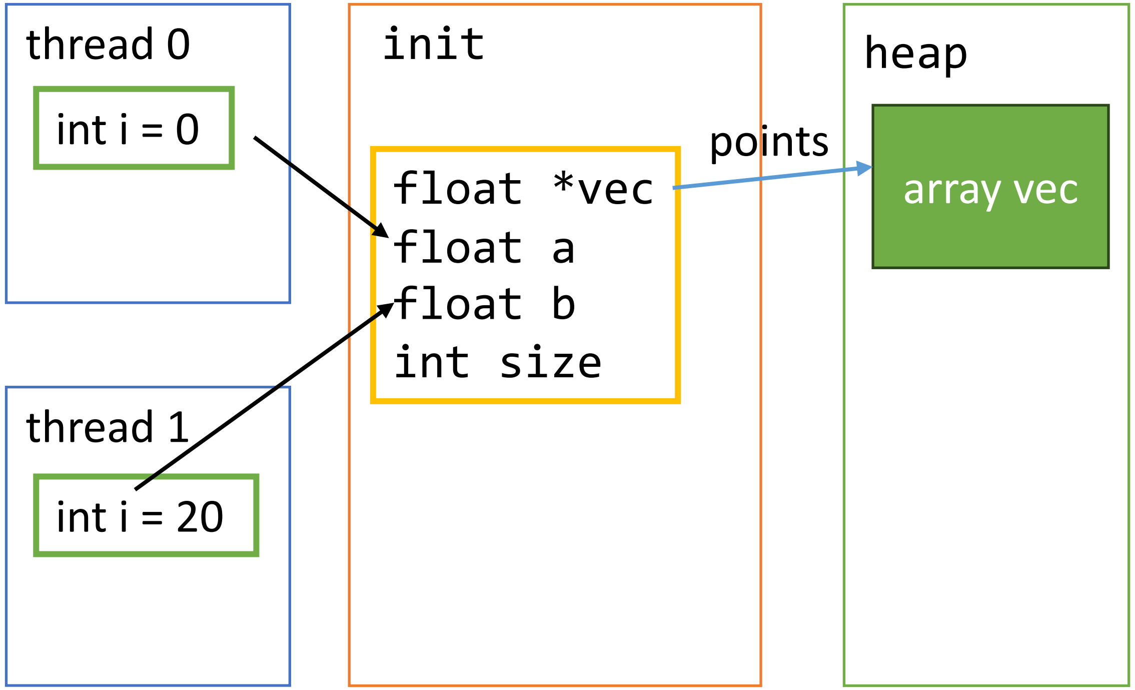
The Inner Loop of the assembly code

```
.LBB2_19: # =>This Inner Loop Header: Depth=1
movss xmm0, dword ptr [rbx] # xmm0 = mem[0],zero,zero,zero
mulss xmm0, dword ptr [r14]
movss dword ptr [rax], xmm0
movss xmm0, dword ptr [rbx] # xmm0 = mem[0],zero,zero,zero
mulss xmm0, dword ptr [r14]
movss dword ptr [rax], xmm0
movss xmm0, dword ptr [rbx] # xmm0 = mem[0],zero,zero,zero
mulss xmm0, dword ptr [r14]
movss dword ptr [rax], xmm0
movss xmm0, dword ptr [rbx] # xmm0 = mem[0],zero,zero,zero
mulss xmm0, dword ptr [r14]
movss dword ptr [rax], xmm0
add ebp, -4
jne .LBB2_19
```

Caveat 2: Variable Privatization(Cont’d)

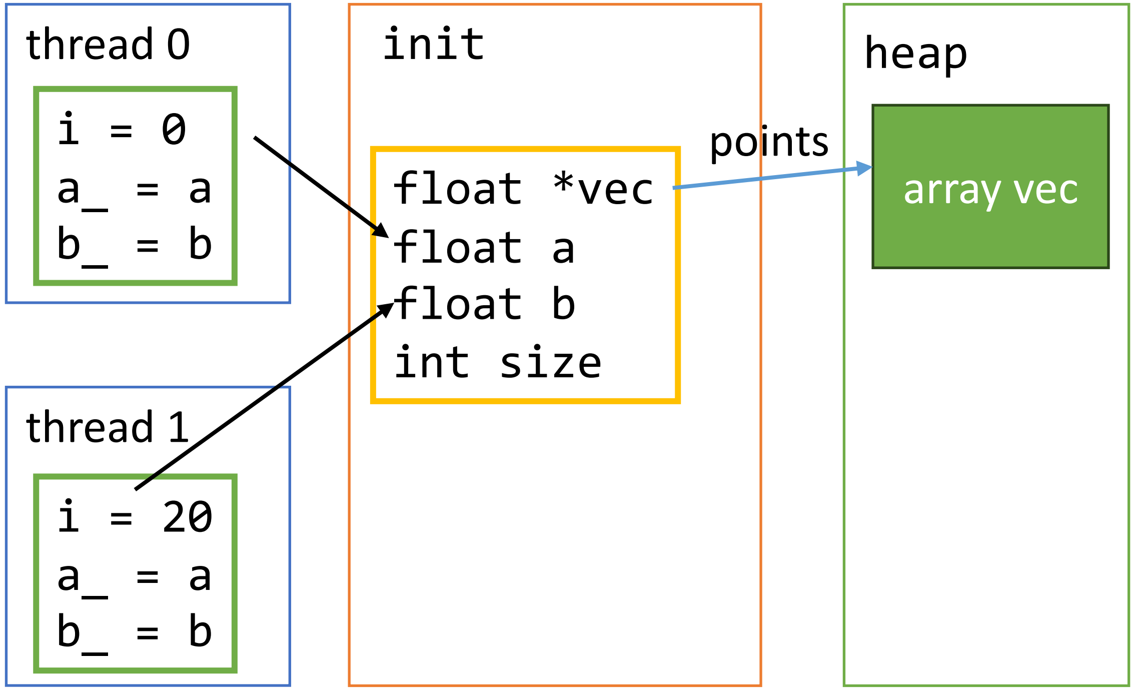
```
#pragma omp parallel for
for (int i = 0; i < size; i++)
  vec[0] = a * b;
```

- According to the API, **a** and **b** are shared variables that reside outside of the thread function.
- When the compiler limits the optimization scope to the outlined function, **a** and **b** are “global variables”.
- Therefore, the outlined function loads **a** and **b** for every use.
- [[Note]] This happens even if we change `vec[0]` to `vec[i]`



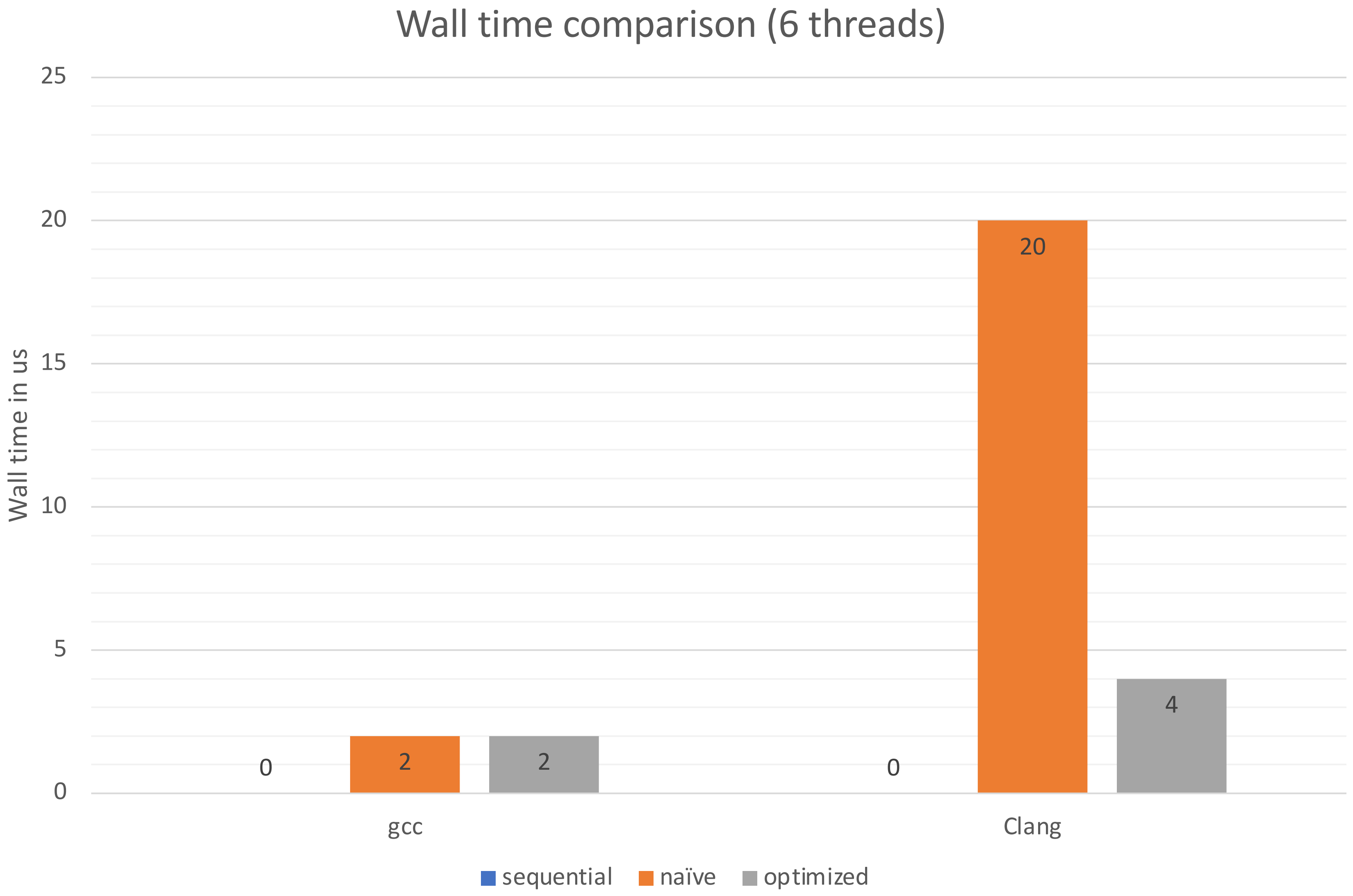
Caveat 2: Variable Privatization(Cont’d)

```
void opt_init(float a, float b,
float* vec, int size) {
#pragma omp parallel
  {
  const float a_ = a, b_ = b;
#pragma omp for
  for (int i = 0; i < size; i++)
    vec[0] = level0(a_, b_);
  }
}
```



- Solution: Manually add private copies of **a** and **b**.

Caveat 2: Variable Privatization(Cont’d)



Summary & Final Notes

- Rule of Thumb: Do not parallelize what you don’t need to.
- If you need parallel processing, OpenMP is a powerful API that parallelizes your workload conveniently.
- Internally, it inserts runtime calls and outlines parallel regions.
    - This makes optimizing codes across OpenMP directives challenging for compilers.
- ***Inspect the code generated by compilers***

Miscellaneous

- System configuration
  - Intel(R) Core(TM) i5-10400 CPU @ 2.90GHz
  - Samsung DDR4 2666MT/s 8GB x2
  - Ubuntu 22.04, GCC=11.3.0, Clang=14.0.0
- All parallel experiments are performed with following command:
  - `numactl --physcpubind=0-5 --membind=0 ./main -t 6 -m $mode -n 1000`
  - `size = 1 Million (2 ** 20)`
- This slide and associated codes are available on the following repository:
  - https://github.com/sailor1493/omapp
- Acknowledgement
  - Most of the code structure come from SHPC TA's implementation.
  - Thanks to Professor Lee., and TAs!