C/C++ Programming Tools

Parallel Programming Lab2-1
Oct 20 2022

Compilers

GCC family

- `gcc`
- `g++`

LLVM/Clang family

- `clang`
- `clang++`

GCC vs Clang

- They have mostly the same usage
  - Flags (options) supported on one should mostly work out-of-the-box on the other
- Initially Clang put a lot of effort on providing better error messages, and but now GCC has catched on
- We suggest trying to use Clang if you cannot understand GCC's error messages, and vice versa.
- Also they sometimes produce faster executables than each other.

MPI Compiler Wrapper

`mpicc` calls `gcc` under the hood

`mpicxx` calls `g++` under the hood

To use `mpicc` with `clang`, call `mpicc -cc=clang`

To use `mpicxx` with `clang++`, call `mpicxx -cxx=clang++`

GCC/Clang optimization flags

- `-O2`

  Turns on most optimizations

- `-O3`

  Turns on all optimizations in `-O2`, plus optimizations that trade off code size for execution time

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

GCC/Clang other flags

- `-l`*`XXX`*
  Link the *`XXX`* library.
  For example, `-lm` links the math library `libm.so`
  `mpicc` automatically adds `-lmpi` for you
- `-o` *`XXX`*
  Set the output executable filename to *`XXX`*
- `-g`
  Generate debug information

1. Makefile

Makefile: a very brief introduction

Used to tell the `make` command how to build executables from sources.

Makefile

- We acknowledge make is not a good build system
- But it's most commonly used / known
- So as an unfortunate outcome, we still use Makefiles
- We ask you to submit Makefiles in homeworks because we encourage you to try different compiler options

Example

- Makefile rule format

```
target: requirements
	command
```

- `.PHONY` : specify phony target

```
$ make (= make all)
gcc -o hello -O3 hello.c
gcc -o world -O3 world.c

$ make clean
rm -f hello world
```

```
.PHONY: all
all: hello world

hello: hello.c
	gcc -o hello -O3 hello.c

world: world.c
	gcc -o world -O3 world.c

.PHONY: clean
clean:
	rm -f hello world
```

Example

- Variable
- `%` : stem part
- `$@` : the target
- `$^` : all requirements
- `$<` : the first requirement

```
TARGETS = hello world

.PHONY: all
all: $(TARGETS)

%: %.c
    gcc -o $@ -O3 $<

.PHONY: clean
clean:
    rm -f $(TARGETS)
```

Example

```
CC = gcc
CFLAGS = -O3
TARGETS = hello world

.PHONY: all
all: $(TARGETS)

%: %.c
    $(CC) -o $@ $(CFLAGS) $<

.PHONY: clean
clean:
    rm -f $(TARGETS)
```

Example

```
CC = gcc
CFLAGS = -O3
TARGETS = hello world

.PHONY: all
all: $(TARGETS)

.PHONY: clean
clean:
    rm -f $(TARGETS)
```

2. Debugging Tools

Debugging Tools

- Turn on compiler warnings
- AddressSanitizer
- Clang static analyzer
  - Study by yourself
- gdb
  - Using this in parallel environments is super complex so we're not going to cover it

Turn on compiler warnings

Just add `-Wall` to the list of compiler flags

Compiler warnings off vs on

```
afg@apollo31 ~> cat a.cc
#include <stdio.h>
int main() {
        int a;
        for (int i = 0; i < a; i++) {
                printf("%d\n", i);
        }
}
afg@apollo31 ~> gcc a.cc
afg@apollo31 ~> gcc -Wall a.cc
a.cc: In function ‘int main()’:
a.cc:4:20: warning: ‘a’ is used uninitialized in this function [-Wuninitialized]
    4 |  for (int i = 0; i < a; i++) {
      |                    ~~^~~
```

AddressSanitizer

Asks the compiler to inject code to check memory access during execution.

https://github.com/google/sanitizers/wiki/AddressSanitizer

AddressSanitizer

Add `-fsanitize=address -g` to the compiler. Then run your code as usual. Asan will crash your program if something wrong happened.

AddressSanitizer works with either GCC or Clang.

Enabling Asan would harm performance! Only use it for debugging, don't submit your code with Asan enabled.

```c
#include <stdio.h>

int max(int* a, int n) {
    if (n == 0) return 0;
    int result = a[0];
    for (int i = 1; i < n; i++) {
        if (a[i] > result) result = a[i];
    }
    return result;
}

int main() {
    int arr[] = {5, 4, 3, 9, 1, 2};
    printf("The max value in the array is: %d\n", max(arr, sizeof(arr)));
}
```

```
afg@apollo31 ~> clang -fsanitize=address -g max.c
afg@apollo31 ~> ./a.out
=================================================================
==28619==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffea1bcc6d
8 at pc 0x5615cf424607 bp 0x7ffea1bcc630 sp 0x7ffea1bcc628
READ of size 4 at 0x7ffea1bcc6d8 thread T0
    #0 0x5615cf424606 in max /home/afg/max.c:7:7
    #1 0x5615cf4247a7 in main /home/afg/max.c:14:48
    #2 0x7f7704162ee2 in __libc_start_main (/usr/lib/libc.so.6+0x26ee2)
    #3 0x5615cf30f09d in _start (/home/afg/a.out+0x1f09d)
```

```
afg@apollo31 ~> clang -fsanitize=address -g max.c
afg@apollo31 ~> ./a.out
=================================================================
==28619==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffea1bcc6d
8 at pc 0x5615cf424607 bp 0x7ffea1bcc630 sp 0x7ffea1bcc628
READ of size 4 at 0x7ffea1bcc6d8 thread T0
    #0 0x5615cf424606 in max /home/afg/max.c:7:7
    #1 0x5615cf4247a7 in main /home/afg/max.c:14:48
    #2 0x7f7704162ee2 in __libc_start_main (/usr/lib/libc.so.6+0x26ee2)
    #3 0x5615cf30f09d in _start (/home/afg/a.out+0x1f09d)
```

```c
#include <stdio.h>

int max(int* a, int n) {
    if (n == 0) return 0;
    int result = a[0];
    for (int i = 1; i < n; i++) {
        if (a[i] > result) result = a[i];
    }
    return result;
}

int main() {
    int arr[] = {5, 4, 3, 9, 1, 2};
    printf("The max value in the array is: %d\n", max(arr, sizeof(arr)));
}
```

sizeof(arr) / sizeof(int)

Study by yourself

Clang static analyzer

Perform more complex compile-time static analysis than `-Wall`

Clang static analyzer: usage

Use one of the following

1. Compile with `clang --analyze`
2. Use `scan-build -o` *`outputpath`* `make` instead of `make`

Then, you can either:

1. Run `scan-view --host 0.0.0.0 --allow-all-host` *`outputpath`*`/XXX`
2. Or download `outputpath/XXX` and view locally

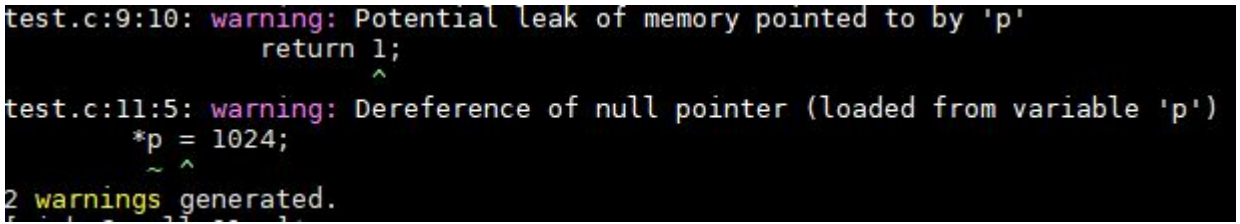
```
1  #include <stdlib.h>
2
3  int main() {
4          int * p = malloc(sizeof(int));
5          if (p != NULL) {
6                  return 1;
7          }
8          *p = 1024;
9          free(p);
10         return 0;
11 }
```

1 'p' initialized here →

2 ← Assuming 'p' is equal to NULL →

3 ← Taking false branch →

4 ← Dereference of null pointer (loaded from variable 'p')


test.c:9:10: warning: Potential leak of memory pointed to by 'p'
return 1;
^
test.c:11:5: warning: Dereference of null pointer (loaded from variable 'p')
*p = 1024;
~ ^
2 warnings generated.