รายงานวิชา Operating Systems

จัดทำโดย กฤติวิทย์ คำประดำ 6610501963

Parallel : โครงสร้างไฟล์ก่อน การ compile เป็น ดังรูป

```
.
├── BUILD.md
├── Makefile
├── compile_commands.json
├── include
│   └── config.h
├── parallel.cpp
└── parallel_demo.ipynb
```

โดยที่หลังจากการ compile (พิมพ์คำสั่ง make) จะทำการ สร้าง folder build ดังรูป

```
.
├── BUILD.md
├── Makefile
├── build
│   ├── main
│   └── parallel.o
├── compile_commands.json
├── include
│   └── config.h
├── parallel.cpp
└── parallel_demo.ipynb
```

จากนั้นก็สารมารถ กด รันได้โดยใช้ คำสั่ง ./build/main หรือ ใช้คำสั่ง make run ก็จะสามารถรันออกมา โดยจะทำการ ทดสอบตาม ไฟล์ include/config.h

```
1   #include <stdint.h>
2
3   int number_test_size = 3;
4   uint64_t numbers[3] = {
5       (uint64_t)1<<30,
6       (uint64_t)1<<45,
7       (uint64_t)1<<60,
8   };
9   int thread_test_size = 4;
10  int num_threads[4] = {1, 2, 4, 8};
11  int num_benchmark = 10;
```

โดย จะแบ่งเป็น 3 ค่าหลักๆ นั่นคือ ขนาดของตัวเลขที่จะทำการแยกตัวประกอบ กับ จำนวณ thread ที่จะใช้ คำนวณ ซึ่งในที่นี้ จะต้องมีการ test ตัวเลข ทั้งหมด 3 ค่า และ จำนวณ thread 4 ค่า รวมเป็น

ต้อง test 12 ครั้ง(4 x 3) และ ค่า สุดท้าย num_benchmark คือการที่ เราจะรัน ทดสอบกี่รอบ ในที่นี้คือ 10 หรือก็คือเราต้องรันโปรแกรมทั้งหมด 120 รอบ

ส่วนต่อมา คือ การแยก ตัวประกอบ นั่นคือ ฟังก์ชันที่แยกตัวประกอบ ซึ่งจะแบ่ง เป็น 2 ส่วน นั่นคือ ส่วนของ serial กับส่วนของ parallel

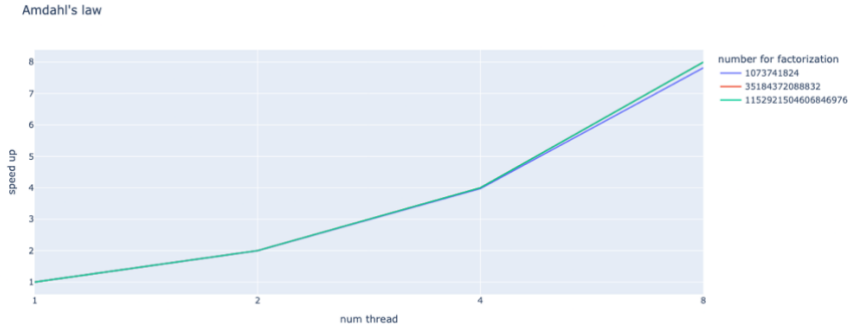
```
31  void bench_mark(uint64_t number, int thread_size, std::vector<double> &result_serial, std::vector<double> &result_parallel, int i){
32      clock_t start, end;
33
34      start = clock();
35      uint64_t sqrt_n = (uint64_t)sqrt((double)number);
36      std::vector<uint64_t> factor;
37      end = clock();
38      result_serial[i] = ((double) (end - start)) / CLOCKS_PER_SEC;
39
40      start = clock();
41      find_factor_pairs(number, sqrt_n, thread_size, factor);
42      end = clock();
43      result_parallel[i] = ((double) (end - start)) / CLOCKS_PER_SEC;
44  }
45
12  void find_factor_pairs(uint64_t number, uint64_t sqrt_n, int thread_size, std::vector<uint64_t> &factor) {
13      omp_set_num_threads(thread_size);
14      #pragma omp parallel
15      {
16          std::vector<uint64_t> local_factor;
17          #pragma omp for nowait
18          for (uint64_t i = 1; i <= sqrt_n; i++) {
19              if (number % i == 0){
20                  local_factor.push_back(i);
21              }
22          }
23          #pragma omp critical
24          {
25              factor.insert(factor.end(), local_factor.begin(),local_factor.end());
26          }
27      }
28  }
```

จะเห็นว่า ต้องรับค่า 1.จำนวณที่จะแยกตัวประกอบ 2.จำนวณ thread ส่วน 3,4,5 เป็นการบันทึกผล

จะเห็นว่า ขั้นแรกเราจะ process ใน ส่วนของ serial ก่อน ค่าต่างๆ แล้วก็เก็บค่า ไว้

จากนั้น ก็จะทำการ คำนวณ แยก factor

จะเห็นว่า ส่วนนี้จะเริ่มเป็นส่วนที่ไม่ใช่ parallel เริ่มจาก รับ number แล้วจากนั้น ใช้ คำสั่ง ของ openmp ในตั้งค่าว่าจะรัน กี่ thread omp_set_num_threads(thread_size) จากนั้น ก็จะทำการใช้คำสั่งของ openmp ในการแบ่งให้แต่ละ process ทำงาน แยกกันโดยไม่ต้อรอ process อื่น ซึ่ง ตรงส่วน นี้ openmp จะจัดการให้โดยอัตโนมัติ จากนั้น ก็จะทำการจับเวลาและบันทึกค่าไว้ซึ่งจะแสดงใน colab ได้ผลดังนี้



1. รับ input เป็น ขนาดของ memory ในหน่วย MB จากนั้นก็ทำการ แปลงให้อยู่ในหน่วย bytes จากนั้น จองพื้นที่หน่วยความจำที่มีปริมาณมากจนสังเกตได้ ใช้หน่วยความจำนั้น สร้างอาร์เรย์

```
16          int memsize;
17          if (argc > 1) {
18              memsize = atoi(argv[1]) * 1024 * 1024; // Convert MB to bytes
19          } else {
20              memsize = 50 * 1024 * 1024; // default 50 MB
21          }
22          printf("before allocate memory: ");
23          show_mem();
24
25          if (memsize % sizeof(int) != 0){
26              printf("memsize doesn't fit in sizeof int\n");
27              return 1;
28          }
29          int num_element = memsize / sizeof(int);
30          int *a = (int *)malloc(memsize);
31          if (!a){
32              printf("cannot use malloc\n");
33              return 1;
34          }
```

2. Fork ออกมาเป็น process ใหม่

```
43          pid_t child_id = fork();
44          if (child_id < 0){
45              free(a);
46              printf("cannot fork child process\n"
47              return 1;
48          }
```

3. ลอง เช็ค Resident Set Size ก่อน ระหว่างเปลี่ยน และหลังจาก เปลี่ยน memory ที่ procress หลัก allocate มา

```
printf("child use before write a memory: ");
show_mem();

// Write to the array to trigger COW
for (int i=0; i < num_element/2; i++){
    a[i] = 2;
}
printf("child use after write half of memory: ");
show_mem();

for (int i=num_element/2; i < num_element; i++){
    a[i] = 2;
}
printf("child use after write full of memory: ");
show_mem();
exit(0);
```

4. ไม่ว่า จะเปลี่ยน memory size เป็นเท่าไหร่ ก็จะยัง ได้ผลลัพธ์ที่คล้ายคลึงกัน

```
build/main 50
before allocate memory: 1 MB
after allocate and write memory: 51 MB
child use before write a memory: 0 MB
child use after write half of memory: 25 MB
child use after write full of memory: 50 MB
parent use: 51 MB
child exit status: 0

build/main 60
before allocate memory: 1 MB
after allocate and write memory: 61 MB
child use before write a memory: 0 MB
child use after write half of memory: 30 MB
child use after write full of memory: 60 MB
parent use: 61 MB
child exit status: 0

build/main 70
before allocate memory: 1 MB
after allocate and write memory: 71 MB
child use before write a memory: 0 MB
child use after write half of memory: 35 MB
child use after write full of memory: 70 MB
parent use: 71 MB
child exit status: 0
```

Credit:

https://cppreference.com/w/cpp/container/vector/insert.html

https://www.openmp.org/wp-content/uploads/SC17-Kale-LoopSchedforOMP_BoothTalk.pdf

https://plotly.com/python/line-charts/

https://man7.org/linux/man-pages/man2/getrusage.2.html