Tối ưu hoá hiệu năng chương trình cho C/C++*

Phạm Ngọc Tân† 19520925@gm.uit.edu.vn

Võ Khánh An† 19520007@gm.uit.edu.vn

Tháng 06/2021

Abstract

Ngày nay, thế giới ngày càng phát triển nhanh chóng và đầy tính cạnh tranh, hiệu năng của một chương trình cũng quan trọng đối với người dùng không thua kém gì các tính năng mà chương trình cung cấp. Trong bài báo cáo này, chúng tôi sẽ giới thiệu một số nguyên lý cho phép tối ưu hoá C/C++. Từ đó giúp cho việc thiết kế một chương trình C/C++ chạy nhanh hơn và tiêu tốn ít tài nguyên hơn trên bất kỳ máy tính nào. Bên cạnh đó, chúng tôi cũng sẽ tiến hành thử nghiệm trên một số ví dụ để minh hoạ cho việc làm thế nào áp dụng những nguyên lý này nhằm cải thiện đoạn code hiện tại để có thể đáp ứng được nhu cầu của người dùng.

1 Giới thiệu

Bài báo cáo này tập trung trình bày về về tối ưu hoá, cụ thể là tối ưu hoá các chương trình C/C++. Một số kỹ thuật trong bài báo này có thể áp dụng trong các ngôn ngữ lập trình khác nhưng chúng tôi không thể đảm bảo điều này bởi vì một số kỹ thuật có thể hiệu quả trên C/C++ nhưng lại không tạo ra ảnh hưởng lớn hoặc thậm chí là không thể thực hiện được trên các ngôn ngữ khác.

Về mặt định nghĩa, tối ưu hoá là một hoạt động viết code [2]. Trong quy trình phát triển phần mềm truyền thống, tối ưu hoá chỉ diễn ra sau khi đoạn code đã được hoàn thành và diễn ra trong suốt giai đoạn tích hợp và thử nghiệm của một dự án.

Mục tiêu của tối ưu hoá là cải thiện một chương trình đã đúng để nó có thể đáp ứng nhu cầu của người dùng về mặt tốc độ, thông lượng, bộ nhớ, tiêu thụ điện năng,... Do đó, tối ưu hoá là một công việc rất quan trọng trong quá trình phát triển không thua kém gì việc tạo ra các tính năng. Một chương trình có hiệu năng tệ đến mức không thể chấp nhận được có thể được xếp vào nhóm những vấn đề trầm trọng của chương trình tương đương với bug và thiếu tính năng.

Một điểm khác biệt quan trọng giữa gỡ lỗi (debug) và tinh chỉnh hiệu năng chính là việc hiệu năng là một biến liên tục. Một tính năng có thể đã được code hoặc không. Một lỗi có thể xuất hiện hoặc không. Tuy nhiên, hiệu năng có thể rất tốt hoặc rất tệ hoặc đâu đó nằm giữa rất tốt và rất tệ. Tối ưu hoá là một quá trình lặp đi lặp lại, khi phần chậm nhất trong chương trình được cải thiện thì một phận chậm nhất mới lại xuất hiện.

Khi viết một chương trình bằng C/C++ hoặc trong bất kỳ ngôn ngữ lập trình nào, mục tiêu đầu tiên và quan trọng nhất chính là việc làm cho chương trình có thể *thực thi được* và *chính xác*. Sau đó chúng ta mới bắt đầu suy nghĩ đến chuyện tối ưu hoá. Có 3 điều mà chúng ta cần xem xét khi tối ưu hoá là: tính bảo mật của chương trình, bộ nhớ tiêu tốn và tốc độ của chương trình. Trong phạm vi bài này, chúng tôi chỉ tập trung vào việc tối ưu hoá về mặt bộ nhớ và tốc độ.

2 Một số vấn đề khi tối ưu hoá trong C/C++

Đôi khi chúng ta cần phải đánh đổi một số thứ khi thực hiện việc tối ưu hoá. Chẳng hạn như việc tối ưu hoá về mặt tốc độ bằng một số kỹ thuật sẽ làm tiêu tốn bộ nhớ khổng lồ. Điều này rõ ràng dẫn đến việc xung đột giữa 2 mục đích tối ưu hoá khác nhau là tối ưu hoá về mặt tốc độ và tối ưu hoá về mặt

*Đây là báo cáo đồ án môn học Nguyên lý Phương pháp Lập trình - CS111.L21.KHCL tại trường Đại học Công nghệ Thông tin - ĐHQG HCM

bộ nhớ. Ngoài ra, một số nỗ lực tối ưu hoá bằng những kỹ thuật kém cũng mang lại một hậu quả khôn lường. Các lập trình viên không thể mang tối ưu hoá ra làm một cái cớ để vi phạm các nguyên tắc trong việc phát triển phần mềm. Bên cạnh đó, một số nỗ lực tối ưu hoá có thể dẫn đến thất bại và khiến cho chương trình có hiệu năng thậm chí kém hơn cả lúc chưa tối ưu.

Cách tối ưu hoá tốt nhất chính là hãy viết một ra đoạn code đơn giản, nhanh gọn để giải quyết bài toán đã được đề ra. Sau đó, chúng ta sẽ quay lại tìm những đoạn code kém và tiến hành thực hiện tối ưu. Tuy nhiên, cố gắng luyện tập viết code một cách tối ưu như một thói quen từ những thứ nhỏ nhất có thể giúp các lập trình viên tiết kiệm được rất nhiều thời gian trong quá trình tối ưu.

Luật Ahmdal

Khi chúng ta đã có được nền tảng về tối ưu hóa và chúng ta bắt đầu công việc tối ưu hóa đoạn code hay chương trình của chúng ta, thì ắt hẳn sẽ đâu đó tồn tại câu hỏi rằng "làm sao để chúng ta có thể đo đạc được hiệu suất chương trình của chúng ta hậu tối ưu hóa?", "làm sao ta biết được liệu chương trình mới của chúng ta có hiệu suất tốt hơn chương trình cũ hay không?". Và đó là khi luật Ahmdal ra đời.

Luật Ahmdal là quy tắc đo đạc hiệu suất gia tăng của chương trình. Luật này chỉ ra rằng: lượng tốc độ được gia tăng sau khi chúng ta tiến hành tối ưu hoá chương trình được tính bằng cách lấy tốc độ của đoạn chương trình cũ chia cho tốc độ của đoạn chương trình mới. Hay theo một cách toán học, luật Ahmdal phát biểu rằng tốc độ của chương trình sau khi được tối ưu hoá được mô hình hoá bằng công thức như sau:

$$\text{Tỉ suất tăng} = \frac{\text{thời gian thực thi}_{\text{cũ}}}{\text{thời gian thực thi}_{\text{mới}}} = \frac{1}{(1 - f_{cost}) + \frac{f_{cost}}{f_{speedup}}} \tag{1}$$

Trong đó:

- f_{cost}: tỉ lệ thời gian thực thi của hàm f so với tổng thời gian thực thi chương trình
- $f_{speedup}$: tỉ suất tăng của hàm f

Để có một cái nhìn tổng quan hơn, chúng ta hãy lấy một ví dụ đơn giản như sau: giả sử chúng ta tối ưu một hàm chiếm 40% tổng thời gian thực thi của toàn bộ chương trình, và hàm đó có tốc độ thực thi nhanh gấp đôi tốc độ thực thi ban đầu.

Theo như luật Ahmdal, ta có tỉ lệ thời gian thực thi của chương trình mới so với chương trình ban đầu là:

$$\begin{aligned}\text{Tỉ suất tăng} &= \frac{1}{(1 - f_{cost}) + \frac{f_{cost}}{f_{speedup}}} \\ &= \frac{1}{(1 - 0.4) + \frac{0.4}{2}} = 1.25\end{aligned}$$

Qua tính toán, ta có thể thấy được tỉ suất tăng giữa đoạn chương trình mới và đoạn chương trình cũ là 1.25, tương đương với việc chương trình mới của ta chạy nhanh hơn chương trình cũ 25%.

3 Các mô hình tối ưu hoá hiệu năng chương trình C/C++

Sự phong phú của các tính năng trong C/C++ cung cấp cho chúng ta hàng loạt các lựa chọn khác nhau để triển khai một dự án. Việc này gúp chúng ta có thể tinh chỉnh các chương trình C/C++ để đáp ứng các yêu cầu về hiệu năng từ người dùng. C/C++ có một số *điểm nóng* trong việc tối ưu như lời gọi hàm, cấp phát bộ nhớ và vòng lặp. Sau đây sẽ là các mô hình tối ưu hoá trong C/C++.

3.1 Sử dụng các cấu trúc dữ liệu phù hợp

Mặc dù máy tính đã có thể lý hàng triệu phép tính mỗi giây, nhưng khi một bài toán trở nên phức tạp, cách tổ chức dữ liệu trở nên vô cùng quan trọng.

Để làm rõ điều này, chúng tôi xin đưa ra một ví dụ như sau: Chúng ta cần đến thư viện để tìm kiếm một quyển sách với lĩnh vực nào đó. Các cuốn sách được xếp theo lĩnh vực. Trong mỗi lĩnh vực, sách lại được xếp theo tên tác giả. Nhờ đó, việc lấy và trả sách từ kệ sách trở nên tiện lợi và nhanh chóng hơn.

Hãy thử tưởng tượng nếu thay vì sách được tổ chức thành từng lĩnh vực, tên tác giả như trên mà chất thành từng đống lộn xộn trong thư viện. Việc tìm được quyển sách chúng ta cần có thể mất đến hàng giờ, thậm chí rất nhiều ngày. Tương tự như vậy, một chương trình không thể vận hành hiệu quả nếu không được lưu trữ một cách phù hợp. Dưới đây, chúng ta sẽ tìm hiểu một số cấu trúc dữ liệu lưu trữ và đưa ra những so sánh.

3.1.1 Mảng

Mảng [1] là một cấu trúc dữ liệu cực kỳ đơn giản và có thể xem như một danh sách với chiều dài cố định. Mảng sẽ thích hợp cho các tình huống mà chúng ta biết trước được số lượng phần tử hoặc có thể xác định được khi chạy chương trình.

```
// Việc xoá 1 phần tử trong mảng sẽ mất chi phí là O(n)
int deleteElement(int arr[], int n, int key {
    int pos = findElement(arr, n, key);
    if (pos == - 1) {
        cout << "Element not found";
        return n;
    }

    for (int i = pos; i < n - 1; i++)
        arr[i] = arr[i + 1];

    return n - 1;
}
```

Một trong những sức mạnh khác của mảng chính là ta có thể truy cập các phần tử của mảng một cách ngẫu nhiên bằng chỉ số trong $O(1)$. Tuy nhiên, cũng vì lý do này, việc tăng kích thước mảng hay thêm/xoá phần tử vào một vị trí bất kỳ của mảng có độ phức tạp lên đến $O(N)$.

3.1.2 Danh sách liên kết

Danh sách liên kết là một cấu trúc dữ liệu có thể giữ một số lượng phần tử tuỳ ý và dễ dàng thay đổi kích thước, cũng như dễ dàng bỏ đi các phần tử nó đang nắm giữ.

```
struct ListNode {
    int data;
    ListNode* nextNode;
};

ListNode* firstNode;

//Chèn node trong danh sách liên kết chỉ mất O(1)
ListNode* newNode = new ListNode();
newNode->nextNode = firstNode;
firstNode = newNode;

//Duyệt qua toàn bộ các phần tử trong danh sách liên kết
ListNode* curNode = firstNode;
while (curNode != NULL) {
   cout << curNode->data << endl;
   curNode = curNode->nextNode;
}
```

Với thiết kế của nó, một danh sách liên kết thích hợp để lưu trữ dữ liệu khi chưa biết trước được số lượng phần tử hoặc các phần tử thường xuyên thay đổi. Tuy vậy, chúng ta không thể truy cập một cách ngẫu nhiên các phần tử của danh sách liên kết. Để tìm kiếm một giá trị, ta phải bắt đầu tại phần tử đầu tiên và duyệt tuần tự qua các phần tử cho tới khi bắt gặp được giá trị mà mình cần tìm kiếm. Để chèn một nút vào danh sách liên kết, bạn cũng phải thực hiện tương tự. Độ phức tạp của cả 2 thao tác này là $O(N)$. Tuy nhiên, nếu ta biết được con trỏ trỏ đến phần tử cần xóa, thì độ phức tạp

chỉ là $O(1)$. Dễ dàng nhận thấy, thao tác tìm kiếm và chèn trong danh sách liên kết không thật sự hiệu quả.

Mảng và danh sách liên kết là 2 cấu trúc dữ liệu nền tảng cho tất cả các loại cấu trúc dữ liệu khác. Chúng được dùng khi chúng ta muốn lưu nhiều dữ liệu (thường cùng kiểu dữ liệu). Bảng dưới đây so sánh các thao tác về mảng và danh sách liên kết:

	Mảng	**Danh sách liên kết**
Bộ nhớ	Cố định (cần biết trước số phần tử)	Có thể tăng giảm tùy ý
Thêm/Xóa 1 phần tử	$O(N)$	$O(1)$, giả sử biết con trỏ tới phần tử đó
Tìm kiếm 1 phần tử	$O(N)$	$O(N)$
Truy cập phần tử	$O(1)$	$O(N)$
Khác	Ít bộ nhớ hơn	

Có thể thấy việc các thao tác trên mảng và kiểu dữ liệu có sự khác biệt nhau khá lớn về mặt chi phí. Do đó ở từng bài toán khác nhau, chúng ta phải cân nhắc sử dụng cấu trúc dữ liệu nào để đạt được hiệu quả tối đa.

3.2 Tối ưu hoá các biến cấp phát động

Bên cạnh việc sử dụng các thuật toán có hiệu năng kém, việc sử dụng một cách ngây thơ các biến được cấp phát động cũng là tác nhân giết chết hiệu năng một cách nhanh chóng trong các chương trình C/C++.

C/C++ cung cấp rất nhiều tính năng giúp chúng ta thực hiện việc cấp phát động như smart pointers, strings,... Điều này giúp việc viết phần mềm trên C/C++ trở nên hiệu quả hơn. Tuy nhiên cũng có không ít những mặt tối khi sử dụng các tính năng này. Trong việc tối ưu hoá cấp phát động, lệnh *new* không nên bị lạm dụng.

Chúng ta không nên lạm dụng các tính năng hữu ích trong C/C++ để dẫn điến việc cấp phát động bừa bãi. Thay vào đó, chúng ta cần loại bỏ các lời gọi không cần thiết, sử dụng các tính năng tương tự nhưng giúp tăng hiệu suất chương trình.

Một kinh nghiệm là loại bỏ các lời gọi vào trình quản lý bộ nhớ khỏi một vòng lặp hoặc hàm thường được gọi. Đây là những cách giúp tăng hiệu suất đáng kể và giúp chúng ta có thể loại bỏ nhiều lời gọi vào bộ nhớ. Dưới đây là một số ví dụ minh hoạ cho việc tối ưu các biến cấp phát động:

3.2.1 Tạo ra các thể hiện tĩnh từ lớp

Trong C/C++, chúng ta có thể tạo ra các đối tượng động từ lớp. Tuy nhiên, hầu hết các đối tượng có thể được tạo tĩnh (nghĩa là không sử dụng *new*). Tuy đơn giản nhưng nhiều lập trình viên lại bỏ quên điều này.

Dưới đây là một ví dụ điển hình cho việc này:

```
MyClass* myInstance = new MyClass("CS111.KHCL.L21", 10);
```

Rõ ràng cách trên tỏ ra khá tốn kém. Thay vào đó, đối tượng nên được sử dụng một cách tĩnh như sau:

```
MyClass myInstance("CS111.KHCL.L21", 10);
```

myInstance trong trường hợp trên sẽ bị huỷ khi việc thực thi rời khỏi khối lệnh chứa dòng lệnh khai báo nó. Nếu chúng ta cần *myInstance* tồn tại lâu hơn, chúng ta có thể khai báo nó trong phạm vi bên ngoài khối lệnh này.

3.2.2 Khai báo các biến động bên ngoài vòng lặp

Vòng lặp được minh hoạ bên dưới tuy nhỏ nhưng lại có một vấn đề lớn. Vấn đề chính là biến *config* được khởi tạo và cấp phát lại trong mỗi lần lặp.

```
for (auto& filename : namelist) {
    std::string config;
    ReadFileXML(filename, config);
    ProcessXML(config);
}
```

Một cách để làm cho vòng lặp này hiệu quả hơn là di chuyển phần khai báo biến *config* ra bên ngoài vòng lặp. Bên trong vòng lặp, *config* đã bị xoá. Thêm vào đó, *clear* không giải phóng bộ đệm có kích thước động bên trong biến *config*, nó chỉ khởi tạo lại giá trị bằng 0. Sau mỗi lần lặp, *config* sẽ không cần phải cấp phát lại.

```
std::string config;
for (auto& filename : namelist) {
    config.clear();
    ReadFileXML(filename, config);
    ProcessXML(config);
}
```

3.2.3 Ngăn cản các lệnh sao chép không mong muốn khi định nghĩa lớp

Không phải đối tượng nào trong chương trình cũng nên được sao chép. Ví dụ, các đối tượng hoạt động như các thực thể (entities) không được sao chép bởi nếu làm vậy sẽ làm mất đi ý nghĩa của chúng.

Nhiều đội tượng hoạt động như một thực thể có không gian trạng thái vô cùng lớn, ví dụ như một vector gồm 1000 strings hoặc một bảng có 1000 ký tự). Một chương trình vô tình tiến hành sao chép một thực thể có thể hoạt động chính xác nhưng chi phí về mặt thời gian có thể khá lớn.

Nếu việc sao chép một đối tượng là tốn kém hoặc không mong muốn, một cách thích hợp để tránh việc này là cấm sao chép. Khai báo copy constructor và toán tử gán trong phần *private* của lớp có khả năng ngăn được việc này. Do vậy, ta sẽ không cần định nghĩa copy constructor và toán tử gán, chỉ cần khai báo là đủ. Ví dụ:

```
class StudentUIT {
private:
    StudentUIT(StudentUIT const&);
    StudentUIT& operator=(StudentUIT const&);
public:
 ...
};
```

Trong C/C++, từ khoá *delete* được thêm vào cuối phần khai báo của copy constructor và toán tử gán cũng có thể đạt được kết quả tương tự như trên. Nó là một ý tưởng hay để xoá constructors trong phần *public* bởi vì trình biên dịch sẽ cung cấp một thông báo lỗi rõ ràng trong trường hợp này:

```
class StudentUIT {
private:
    StudentUIT(StudentUIT const&) = delete;
    StudentUIT& operator=(StudentUIT const&) = delete;
 ...
};
```

Bất kỳ nỗ lực nào để gán thể hiện của một lớp được khai báo theo cách trên hoặc khởi tạo giá trị cho nó hoặc sử dụng nó làm giá trị của một lớp sẽ dẫn đến lỗi biên dịch.

3.3 Tối ưu các lệnh "nóng"

Tối ưu hoá ở cấp độ câu lệnh có thể được hiểu như là quá trình xoá những câu lệnh ra khỏi luồng thực thi. Vấn đề với việc tối ưu hoá ở mức độ câu lệnh là ngoài các lời gọi hàm, không câu lệnh C/C++ nào sử dụng nhiều hơn một hoặc một vài lệnh máy. Việc tập trung vào tối ưu hoá với quy mô nhỏ như vậy thường không tạo ra nhiều bước cải tiến để công sức của chúng ta có giá trị, trừ khi các lập trình viên

có thể tìm ra các yếu tố làm tăng chi phí của câu lệnh khiến nó đủ *nóng* để tối ưu. Những yếu tố này bao thường là vòng lặp và những hàm được gọi nhiều lần.

Tối ưu hoá ở cấp độ câu lệnh có thể tạo ra những cải thiện đáng kể về mặt hiệu suất trên các bộ xử lý nhỏ, đơn giản như các máy tính nhúng vào các thiết bị, đồ chơi,... bởi vì các câu lệnh được nạp trực tiếp từ bộ nhớ và thực thi lần lượt. Tuy nhiên, với các bộ xử lý của máy tính cá nhân (PC) cung cấp rất nhều sự đồng thời ở cấp độ câu lệnh và lưu vào bộ nhớ đệm để tối ưu hoá tạo ra chi phí rất nhỏ. Do đó trên các chương trình được thiết cho PC, tối ưu hoá cấp độ câu lệnh có thể chỉ thích hợp cho các lời gọi hàm hoặc các vòng lặp trong cùng của các chương trình.

Một vấn đề với việc tối ưu hoá ở cấp độ câu lệnh là hiệu quả của việc tối ưu hoá có thể phụ thuộc vào trình biên dịch. Mỗi trình biên dịch có một hoặc nhiều cách để biên dịch một câu lệnh C/C++. Một vài câu lệnh giúp cải thiện hiệu suất trên một trình biên dịch này có thể không tạo ra kết quả như mong muốn trên một trình biên dịch khác, thậm chí có thể làm chậm chương trình. Một thủ thuật giúp cải thiện hiệu suất khi sử dụng GCC có thể không hoạt động trên Visual C. Nghiêm trọng hơn, nếu chúng ta nâng cấp trình biên dịch lên một phiên bản mới thì những đoạn code đã được tối ưu có thể không hoạt động như mong đợi. Đây lại là một lý do giải thích tại sao tối ưu hoá ở cấp độ câu lệnh tỏ ra kém hiệu quả hơn các hình thức tối ưu hoá khác.

3.3.1 Xoá lệnh tốn chi phí ra khỏi vòng lặp

Một vòng lặp có 2 phần: Một khối các câu lệnh được thực thi lặp đi lặp lại và một mệnh đề có điều kiện xác định số vòng lặp. Với các vòng lặp, mệnh đề kiểm tra điều kiện là một điểm cần xem xét đầu tiên trong việc tối ưu. Ví dụ:

```
char s[] = "This is report for CS111";
 ...
for (int i = 0; i < strlen(s); ++i)
    if (s[i] == ' ')
        s[i] = '-';
```

Trong ví dụ trên, việc kiểm tra i $<$ strlen(s) trong vòng lặp for được thực hiện cho mỗi ký tự trong chuỗi. Hàm strlen() có chi phí khá lớn bởi nó duyệt qua đối số là chuỗi của nó để đếm số ký tự của chuỗi đó. Việc này biến độ phức tạp thuật toán trong ví dụ trên từ O(n) thành O(n^2). Đây là một ví dụ điển hình cho việc một vòng lặp bên trong một hàm thuộc thư viện.

Kết quả thực nghiệm cho thấy, 10 triệu lần lặp của vòng lặp trên mất 13238 mili giây.

Có thể thấy giá trị kết thúc của vòng lặp, ở đây là strlen() có thể được tính toán trước sau đó lưu vào một biến để cải thiện hiệu suất. Đoạn code có thể cải thiện như sau:

```
char s[] = "This is report for CS111";
int len = strlen(s)
 ...
for (int i = 0; i < len; ++i)
    if (s[i] == ' ')
        s[i] = '-';
```

Hiệu quả của sự thay đổi này là rất đáng kể bởi vì strlen() tốn rất nhiều chi phí tính toán. Kết quả thực nghiệm cho thấy đoạn code này chỉ tốn mất 541 mili giây.

3.3.2 Xoá đoạn code bất biến ra khỏi vòng lặp

Trong phần trên, chúng ta đã thấy giá trị kết thúc của vòng lặp được lưu vào một biến để sử dụng lại làm tăng hiệu quả. Đó là một kỹ thuật tổng quát hơn kỹ thuật xoá đoạn code bất biến ra khỏi vòng lặp. Một đoạn code được gọi là bất biến đối với một vòng lặp nếu nó không phụ thuộc vào bất kỳ biến nào trong vòng lặp.

```
int i,j,x,a[10];
 ...
for (i=0; i < 10; ++i) {
    j = 100;
    a[i] = i + j * x * x;
}
```

Trong ví dụ bên trên, câu lệnh gán j = 100 và biểu thức con j * x * x là bất biến trong vòng lặp. Ta có thể tiến hành đem những phép tính này ra ngoài như sau:

```
int i,j,x,a[10];
 ...
j = 100;
int tmp = j * x * x;
for (i=0; i<10; ++i) {
    a[i] = i + tmp;
}
```

Kết quả thực nghiệm cho thấy, khi mang các giá trị bất biến ra khỏi vòng lặp thì thời gian chạy chỉ mất 225,892 mili giây so với 234,375 mili giây lúc chưa tối ưu.

3.3.3 Sử dụng ít các toán tử tốn kém chi phí

Theo lý thuyết, những phép toán thao tác trực tiếp trên dãy nhị phân sẽ tốn ít thời gian hơn so với những phép toán khác. May mắn thay, ngôn ngữ C/C++ đã có cơ chế hỗ trợ cho chúng ta ít nhiều về chuyện đó thông qua các toán tử thao tác bit như dịch bit sang trái, dịch bit sang phải, phép xor hay phép and.

Thông thường nhiều lập trình viên sẽ viết như sau:

```
long long mul = 1;
for (int i = 1; i <= 20; ++i){
    mul *= 4;
}
```

Tuy nhiên việc nhân 1 thừa số cho 4 cũng tương đương với việc chúng ta dịch thừa số đó sang trái 2 bit. Vì thế ta có thể viết lại đoạn code đó như sau:

```
long long mul = 1;
for (int i = 1; i <= 20; ++i){
    mul <<= 4;
}
```

Và thông qua thực nghiệm cho thấy chỉ với 1 thay đổi nhỏ mà ta có thể giảm thời gian thực thi chương trình từ 35ms xuống chỉ còn 2ms.

3.4 Sử dụng một thư viện tốt hơn

Thư viện chuẩn C++ (The Standard C++ Template - STL) là một thư viện mạnh mẽ khiến chúng ta có thể bắt với tốc độ gần như được tối ưu so với những thư viện khác. Việc thuần thục STL là một trong những kỹ năng cần phải có của những lập trình viên C++. STL có thể giúp các lập trình viên giảm thời gian lập trình đáng kể nhưng vẫn đảm bảo chất lượng của dự án. Chúng ta sẽ thấy được sức mạnh của STL C++ trong ví dụ dưới đây.

Thư viện chuẩn C (Standard C library) cung cấp hàm qsort() có thể sử dụng để sắp xếp một mảng.

```
void qsort (void* base, size_t num, size_t size,
int (*comparator)(cons void*, const void*));
```

C++ STL cũng cung cấp một hàm có chức năng tương tự là sort().

```
void sort(T first, T last, Compare comp);
```

Dưới đây là một số so sánh 2 hàm sắp xếp của 2 thư viện là STL C++ và Standard C.

3.4.1 Chi tiết bên trong

Như tên của nó, qsort() sử dụng Quick sort để sắp xếp một mảng đã cho. Trong khi đó sort() trong C++ STL sử dụng một thuật toán lai, bao gồm: Quick sort, Heap sort và Insertion sort

3.4.2 Độ phức tạp

C Standard không nói rõ về độ phức tạp của qsort(). Trong khi đó từ phiên bản C++ 11 trở đi, sort() trong STL có độ phức tạp trong trường hợp xấu nhất và trung bình là $O(NlogN)$

3.4.3 Thời gian chạy

STL sort() chạy nhanh hơn C qsort() bởi vì C++ tạo ra một đoạn code được tối ưu cho từng kiểu dữ liệu cụ thể và một hàm so sánh cụ thể.

STL sort() chạy nhanh hơn từ 20% đến 50% thuật toán quick sort được code bằng tay và nhanh hơn từ 250% đến 1000% C qsort(). C là một trong những ngôn ngữ nhanh nhất như qsort() lại quá chậm.

Khi chúng tôi thử sắp xếp 1 triệu phần tử nguyên trên C qsort() tốn mất 0,25 giây. Trong khi đó, C++ sort() chỉ mất 0,09 giây.

3.4.4 Sự linh động

STL sort() có thể hoạt động trên tất cả các kiểu dữ liệu như mảng, vector, deque,... và cả các kiểu dữ liệu người dùng tự định nghĩa. Sự linh hoạt này khó mà đạt được trên C.

3.4.5 An toàn

So với C qsort(), C++ STL sort() là một kiểu dữ liệu an toàn hơn bởi nó không yêu cầu truy cập vào dữ liệu thông các các con trỏ không an toàn giống như là qsort().

3.5 Sử dụng I/O một cách tối ưu hơn

Đọc và ghi dữ liệu là những hoạt động thường xuyên đến mức mà các lập trình viên thường không để ý đến chúng mặc dù đây là những hoạt động tốn không ít chi phí.

Trong thế giới kết nối internet như hiện nay, việc tốc độ đọc ghi dữ liệu hạn chế có thể dẫn đến một độ trễ cực lớn. Một vấn đề khác của I/O là có rất nhiều đoạn code nằm ở giữa chương trình của người dùng và ổ cứng hoặc card mạng. Chí phí của tất cả code này phải được tối ưu để làm I/O đạt hiệu quả tốt nhất có thể.

Việc đọc file và ghi file là vô cùng cần thiết cho nhu cầu lưu trữ thông tin hiện nay. Các kết quả nhập từ màn hình và xuất ra bàn phím chỉ được lưu trên RAM và có thể biến mất ngoài mong muốn. Bên cạnh đó, việc đọc file cũng giúp người dùng không mất công nhập đi nhập lại nếu input là giống nhau. Hơn thế nữa, việc chờ đợi người dùng nhập vào có thể làm tốn chi phí khi CPU phải chờ đợi các tiến trình, đọc file có thể làm tốt việc này hơn rất nhiều.

```
freopen("DoAnCS111.inp", "r", stdin);
freopen("DoAnCS111.out", "w", stdout);
```

Chúng tôi đã tiến thành đọc một file gồm 10000 dòng với 100 lần và kết quả chỉ tiêu tốn 1548 mili giây. Việc ghi một file tương tự cho kết quả là 2110 mili giây.

3.6 Tối ưu hoá tìm kiếm và sắp xếp

Những thuật toán tìm kiếm và sắp xếp tốt hiện nay đều đã được các lập trình viên tối ưu bằng nhiều phương pháp để có được những hàm giúp chúng ta vừa có thể tiết kiệm thời gian lập trình, mà vừa có thể giúp chúng ta tối ưu hóa được những đoạn chương trình cần phải thực thi một cách hiệu quả hơn. C++ có đã hỗ trợ một thư viện <algorithm> để giúp chúng ta mỗi khi ta cần phải sắp xếp và tìm kiếm. Tuy nhiên, trong thư viện này có rất nhiều hàm khác nhau chỉ để phục vụ một thuật toán, và không phải hàm nào cũng có thể mang lại hiệu suất cao cho những người lập trình như chúng ta. Vì thế, chúng ta cần phải biết sử dụng và hiểu được cách các hàm được thực thi để ta có thể áp dụng và có được một hiệu suất tốt nhất cho chương trình của chúng ta.

3.6.1 Tìm kiếm tuần tự

Khi nói đến các giải thuật tìm kiếm, chúng ta không thể nào mà không nhắc đến tìm kiếm tuần tự *(Linear search)*. Và với giải thuật tìm kiếm tuần tự, C++ đã hỗ trợ cho chúng ta qua việc thiết kế sẵn hàm std::find(). Hàm find() trả về con trỏ tới phần tử đầu tiên bằng với giá trị mà chúng ta đang tìm kiếm. std::find() cũng hỗ trợ chúng ta cho việc giới hạn không gian tìm kiếm thông qua tham số first và last để ám chỉ cho đoạn không gian chúng ta muốn tìm kiếm.

Vì hàm std::find() sử dụng giải thuật tìm kiếm tuần tự, cho nên độ phức tạp trung bình của hàm này là $O(N)$

3.6.2 Tìm kiếm nhị phân

Để giải quyết vấn đề về thời gian mà thuật toán tuần tự sinh ra, chúng ta có một giải thuật khác mang lại hiệu quả cao hơn cho các giải pháp tìm kiếm của chúng ta. Đó chính là tìm kiếm nhị phân *Binary search*. Tìm kiếm nhị phân là một chiến lược chia để trị hiệu quả đến mức mà C++ hỗ trợ kha khá hàm cho nhiều mục đích khác nhau trong lập trình.

3.6.2.1 std::binary_search()

Giải thuật của thư viện chuẩn trong C++ - std::binary_search() - trả về giá trị là một biến boolean biểu thị cho kết quả tìm kiếm rằng giá trị cần tìm có tồn tại bên trong không gian tìm kiếm của chúng ta hay không. Nếu đúng thì hàm sẽ trả về giá trị là 1, còn nếu sai thì hàm sẽ trả về giá trị là 0. Và thật kì lạ khi không có một hàm nào khác giống như vậy để trả về vị trí của phần tử cần tìm trong không gian tìm kiếm. Vì thế nên giải thuật thường chỉ được dùng để xác định xem khóa cần tìm có tồn tại bên trong không gian tìm kiếm hay không, và không thể nào đáp ứng yêu cầu của chúng ta khi tìm kiếm vị trí của phần tử cần tìm bên trong không gian tìm kiếm.

3.6.2.2 std::equal_range()

Hàm std::equal_range() bên trong thư viện <algorithm> sẽ trả về cho chúng ta một cặp con trỏ tương ứng với vị trí bắt đầu và vị trí kết thúc của chuỗi con nằm bên trong không gian tìm kiếm $[first; last)$ mà chứa các giá trị bằng với khóa cần tìm.

Nếu như không tồn tại giá trị nào bằng với khóa cần tìm của ta, hàm std::equal_range() sẽ trả về một cặp giá trị bằng nhau *(hay first = last)* để biểu diễn cho việc giá trị trả về là một chuỗi rỗng.

Nếu như 2 cặp giá trị trả về là khác nhau *(tương ứng với first ≠ last)* thì nghĩa là tồn tại ít nhất một giá trị trong không gian tìm kiếm thỏa khóa cần tìm của chúng ta. Theo lý thuyết, trong bài toán mẫu của lập trình viên đưa ra, không tồn tại nhiều hơn một giá trị cần tìm, và giá trị first sẽ là con trỏ trỏ đến giá trị cần tìm của chúng ta.

Tuy nhiên, dù là hàm này được thiết kế dựa trên giải thuật tìm kiếm nhị phân, nhiều thí nghiệm cho thấy: thời gian thực thi của chương trình này tệ hơn rất nhiều so với các hàm khác cùng sử dụng 1 giải thuật, và thậm chí, std::equal_range() còn thực thi trong khoảng thời gian bằng với thời gian thực thi của std::find(), hoặc thậm chí là tệ hơn, khi xét trên cùng một tập dữ liệu đã được sắp xếp.

3.6.2.3 std::upper_bound() và std::lower_bound()

std::upper_bound() và std::lower_bound() trả về con trỏ đầu tiên của không gian tìm kiếm, tương ứng với khóa lần lượt nhỏ hơn hoặc lớn hơn giá trị chúng ta cần tìm. Cả 2 hàm sẽ trả về con trỏ trỏ tới vị trí cuối cùng trong không gian tìm kiếm nếu như không có giá trị nào thỏa khóa mà ta cần tìm.

Thực nghiệm cho thấy rằng cả 2 hàm std::upper_bound() và std::lower_bound() đều có có hiệu suất tốt hơn hàm std::equal_range() lên tới 86% dù cho cùng sử dụng một nguyên lí thuật toán tìm kiếm nhị phân.

3.6.3 Thực nghiệm

Khi lần lượt thực nghiệm các hàm tìm kiếm khác nhau trên cùng một tập dữ liệu đã được sắp xếp gồm 10^6 phần tử bất kì (*tồn tại các phần tử trùng nhau*) trong đoạn $\left[-10^6; 10^6\right]$, và các truy vấn lần lượt là tìm phần tử đầu tiên, tìm phần tử ở giữa, tìm phần tử ở cuối, tìm phần tử bất kì, ta thu được kết quả như sau:

std::find()	std::binary_search()	std::equal_range()	std::upper_bound()	std::lower_bound()
3e-05 ms	3e-06 ms	5e-06 ms	3e-06 ms	2e-06 ms
0.002375 ms	3e-06 ms	5e-06 ms	2e-06 ms	2e-06 ms
0.004737 ms	3e-06 ms	5e-06 ms	3e-06 ms	1e-06 ms
3.5e-05 ms	4e-06 ms	4e-06 ms	2e-06 ms	2e-06 ms

Kết quả thực nghiệm cho ra đúng so với những gì chúng ta đã trình bày ở trên và những lý thuyết nền tảng về thuật toán. std::find() luôn cho ra kết quả xấu nhất so với các hàm tìm kiếm khác, ta có thể hiểu đơn giản rằng độ phức tạp trung bình của giải thuật tìm kiếm tuyến tính là $O(N)$, trong khi đó độ phức tạp trung bình của giải thuật tìm kiếm nhị phân là $O(logN)$

3.7 Tối ưu hoá xử lí đồng thời

Xử lí đồng thời là việc thực thi đồng thời nhiều luồng xử lí cùng 1 lúc. Mục đích của xử lí đồng thời không phải là việc giảm đi số lượng các dòng lệnh thực thi hay số lượng dữ liệu cần phải xử lí trên giây. Mục đích thật sự của xử lí đồng thời chính là cải thiện việc sử dụng các nguồn tài nguyên trên máy tính để có thể giảm đi thời gian thực thi khi thực hiện chương trình.

3.7.1 Xử lí đồng thời có thể làm cho chương trình của chúng ta chạy nhanh hơn như thế nào ?

Xử lí đồng thời cải thiện hiệu suất bằng việc cho phép những thao tác khác nhau của chương trình có thể tiếp tục trong khi những thao tác khác đang đợi được xử lí hoặc đợi để được cấp phát tài nguyên. Điều này cho phép tài nguyên tính toán có thể được sử dụng thường xuyên hơn. Càng nhiều thao tác khác nhau trong chương trình được tiến hành đồng thời thì ta càng sử dụng càng nhiều tài nguyên. Và càng nhiều hoạt động được diễn ra một cách đồng thời thì ta sẽ cải thiện được hiệu suất của chúng ta dựa trên tổng hiệu suất tính toán đến một điểm bão hòa. Và đương nhiên, chúng ta luôn hy vọng có thể đạt được điểm bão hòa gần với 100% nhất.

Nhìn từ góc độ tối ưu hóa, thử thách cho xử lí đồng thời đó chính là việc đi tìm đủ các công việc độc lập để có thể tận dụng tối đa tài nguyên tính toán có sẵn, kể cả khi một vài công việc cần phải có sự can thiệp từ bên ngoài (như thao tác nhập, xuất) hay tài nguyên hiện có.

Xử lí đồng thời có thể được cung cấp cho chương trình thông qua phần cứng máy tính, hệ điều hành, hay các thư viện có sẵn. May mắn thay, ngôn ngữ C cung cấp cho ta các thư viện phù hợp cho việc xử lí đồng thời dựa trên luồng và chia sẻ bộ nhớ. Và chắc chắn đây là cách duy nhất để những chương trình được viết bằng ngôn ngữ C có thể thực thi hệ thống những chương trình mang tính phối hợp như vậy.

Một vài dạng của xử lí đồng thời có thể được biết đến như sau:

3.7.1.1 Time slice

Time slice (*hay còn gọi là quantum time*) là một chức năng của định thời trong hệ điều hành. Trong time slice, hệ điều hành giữ danh sách các chương trình đang thực thi và các công việc hệ thống hiện tại, và cấp phát thời gian cho từng chương trình. Mỗi khi một chương trình ở trạng thái dừng cho một sự kiện hoặc dừng đợi cấp phát tài nguyên, chương trình thường sẽ truy cập vào danh sách những công việc có thể thực thi trong hệ điều hành và chia sẻ tài nguyên xử lí cho các chương trình khác.

Hệ điều hành vừa phải phụ thuộc vào bộ xử lí, vừa phải phụ thuộc vào phần cứng. Tận dụng bộ định thời để có thể ngắt bộ định thời xử lí khi cần thiết. Và đặc biệt hơn nữa, chương trình thực thi bằng ngôn ngữ C không biết được rằng nó đang bị chia sẻ tài nguyên.

3.7.1.2 Tiểu trình

Tiểu trình là các dòng thực thi đồng thời bên trong một quá trình chia sẻ bộ nhớ. Tiểu trình được đồng bộ hóa bằng việc đồng bộ bộ nhớ gốc và giao tiếp liên tiến trình.

Ưu điểm của việc sử dụng tiểu trình thay vì sử dụng tiến trình đó chính là tiểu trình sử dụng ít tài nguyên hơn, cũng như việc tạo ra các tiểu trình và chuyển đổi giữa các tiểu trình sẽ nhanh hơn.

3.7.1.3 Đa xử lí đối xứng

Đa xử lí đối xứng là việc máy tính chứa nhiều đơn vị thực thi cùng thực hiện một đoạn mã máy và có quyền truy cập đến cùng một bộ nhớ vật lý. Các chương trình thực thi hiện nay và các công việc hệ thống có thể chạy trên bất kì đơn vị thực thi nào còn dư tài nguyên, dù cho việc chọn đơn vị thực thi nào có thể ảnh hướng đến hiệu suất toàn cục của chương trình.

Theo lý thuyết, nếu có n đơn vị xử lí, thì tổng thời gian thực thi chương trình của một chương trình tính toán có thể giảm với tỉ lệ là $\frac{1}{n}$.

3.7.1.4 Đa luồng đồng thời

Đa luồng đồng thời là tính năng hoạt động bằng cách tách từng lõi vật lý của bộ xử lý thành các lõi ảo, hay còn gọi là tiểu trình, cho phép mỗi lõi chạy hai dòng lệnh một lúc.

Vì đa luồng đồng thời chạy hai (hay nhiều) tiểu trình trong hai tiến trình độc lập của cùng một lõi vật lý để tăng hiệu năng, nên một tiến trình có thể biết rất nhiều về những gì tiến trình kia đang thực hiện.

3.7.2 Xử lí đồng thời trong ngôn ngữ C

3.7.2.1 Luồng

Ngôn ngữ C cung cấp cho ta thư viện <thread>, và trong đó cung cấp lớp mẫu std::thread. std::thread cho phép chương trình của ta tạo ra các luồng thông qua các tiểu trình có sẵn của hệ điều hành.

std::thread là một lớp để quản lí các tiểu trình của hệ điều hành. Nó cũng cho phép một chương trình truy cập đến tập các hàm lớn hơn trên hệ điều hành mà có thể thực thi trực tiếp trên tiểu trình.

3.7.2.2 Mutex

Ngôn ngữ C cung cấp các thư viện mẫu cho mutex (*loại trừ tương hỗ*) cho những phần trọng yếu. Định nghĩa của các hàm mutex mẫu đủ đơn giản để có thể thành thạo cho nhiều lớp mutex thuần phụ thuộc với hệ điều hành nhất định.

Thư viện <mutex> gồm 4 hàm mutex mẫu như sau:

- std::mutex
  Một hàm khá là dễ và hiệu quả khi hàm này sẽ cố gắng thực hiện busy-wait trước, rồi trở lại với lời gọi hệ thống nếu như nó không được loại trừ nhanh chóng.

- std::timed_mutex
  Hàm std::timed_mutex cho phép sát nhập mutex trong một khoảng thời gian hạn định. Yêu cầu cho công việc này đó chính là việc can thiệp vào hệ điều hành, điều này sẽ làm gia tăng đáng kể độ trễ của hàm mutex này so với std::mutex.

- std::recursive_mutex
  Hàm std::recursive_mutex cho phép một luồng có thêm mutex khi mutex ở bên trong lời gọi của hàm lồng nhau. Hàm này có thể kém hiệu quả vì cần phải đếm mất bao lâu để nó có thể hoàn thành công việc.

- std::recursive_timed_mutex
  Hàm std::recursive_timed_mutex đúng như tên gọi, là một hàm kết hợp tất cả tính chất của 2 hàm bên trên lại với nhau để có được hiệu quả đem lại là cao nhất.

3.7.3 Thực nghiệm

Để có thể biết rõ hơn về hiệu suất của 2 chương trình, ta thiết lập thực nghiệm như sau: viết 2 chương trình cùng thực hiện việc tính tổng các số fibonanci trong đoạn $[1; 45]$, một chương trình sử dụng luồng, còn chương trình kia thì không. Cụ thể ta có:

fibo_nonthread.c

```
long long fibo(int n){
    if (n <= 1)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
int main(int argc, char* argv[]){
    int count = atoi(argv[1]);
    for(int i = 1; i <= count; ++i) {
        sum = fibo(i);
    }
}
```

fibo_thread.c

```
void *runner(void *param) {
        sum = fibonacci((int)param);
        pthread_exit(0);
}

int fibonacci(int x) {
        if (x <= 1) {
                return 1;
        }
        return fibonacci(x - 1) + fibonacci(x - 2);
}
int main(int argc, char *argv[]) {
        int count, i;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        count = atoi(argv[1]);
        for (i = 1; i <= count; i++) {
                pthread_t thread;
                pthread_create(&thread, &attr, runner, (void *)i);
                pthread_join(thread, NULL);
        }
}
```

Thực hiện trong 3 lần, ta có kết quả thực nghiệm như sau. Biết rằng fibo_nonthread.c là đoạn chương trình không sử dụng luồng, còn fibo_thread.c là đoạn chương trình có sử dụng luồng trong tính toán.

fibo_nonthread.c	fibo_thread.c
36.686181 seconds	32.118316 seconds
32.536268 seconds	31.743584 seconds
36.979518 seconds	30.730709 seconds

Kết quả thực nghiệm cho ta thấy được rằng, việc sử dụng luồng trong lập trình giúp chúng ta cải thiện đáng kể thời gian thực thi chương trình.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009. ISBN 0262033844.

[2] K. Guntheroth. *Optimized C++: Proven Techniques for Heightened Performance.* O'Reilly Media, Inc., 1st edition, 2016. ISBN 1491922060.