CẢI TIẾN THỰC THI ĐỘT BIẾN TRONG KIỂM THỬ ĐỘT BIẾN CHO CÁC MÔ HÌNH SIMULINK SỬ DỤNG TÍNH TOÁN SONG SONG

IMPROVING MUTATION EXECUTION IN MUTATION TESTING FOR SIMULINK MODELS USING PARALLEL COMPUTING

Lê Thị Mỹ Hạnh, Khuất Thanh Tùng, Nguyễn Thanh Bình

Trường Đại học Bách khoa, Đại học Đà Nẵng; Email: ltmhanh@dut.udn.vn

Tóm tắt – Kiểm thử đột biến là một chiến lược kiểm thử dựa trên lỗi để đánh giá chất lượng kiểm thử, bằng cách chèn lỗi vào chương trình đang kiểm thử. Kiểm thử đột biến không chỉ cho phép xây dựng các bộ dữ liệu thử chất lượng, nghĩa là có khả năng phát hiện lỗi cao, mà còn có thể dễ dàng được tự động hóa nhằm giảm chi phí. Vì vậy, kiểm thử đột biến là một trong những kỹ thuật kiểm thử hộp trắng được ứng dụng rộng rãi. Tuy nhiên, một trong những hạn chế của kiểm thử đột biến là thời gian thực thi đột biến khá cao, do số lượng đột biến sinh ra nhiều. Trong bài báo này, chúng tôi trình bày giải pháp để cải thiện chi phí thời gian thực thi đột biến trên các mô hình Simulink sử dụng tính toán song song trên các máy tính đa nhân.

Từ khóa – kiểm thử phần mềm, kiểm thử đột biến, chi phí kiểm thử đột biến, Simulink, tính toán song song.

Abstract – Mutation testing is a fault-based testing strategy to measure the quality of testing by inserting faults into the program under test. Mutation testing not only allows the design of good quality tests, i.e. high error detection capability, but can also be easily automated to reduce cost. Hence, mutation testing is one of white-box testing methods popularly applied. One problem that prevents mutation testing from becoming a practical testing technique is the high computational cost of executing enormous number of mutants against a test set. In this paper, we propose an approach to improving the cost of mutation execution for Simulink models using parallel computing on a multicore machine.

Key words – software testing, mutation testing, mutation testing cost, Simulink, parallel computing.

1. Đặt vấn đề

Kiểm thử phần mềm là một hoạt động đóng vai trò rất quan trọng để bảo đảm chất lượng phần mềm và là hoạt động mang tính sống còn trong các dự án sản xuất hoặc gia công phần mềm. Trong đó, với mục đích phát hiện lỗi, hoạt động kiểm thử phần mềm thường phải trải qua các bước: tạo dữ liệu thử, thực thi phần mềm trên dữ liệu thử và quan sát kết quả nhận được. Trong các bước này, bước tạo dữ liệu đóng vai trò quan trọng nhất, bởi vì chúng ta không thể tạo ra mọi dữ liệu từ miền vào của chương trình, mà chúng ta chỉ có thể tạo ra các dữ liệu thử có khả năng phát hiện lỗi cao nhất.

Có một tiêu chuẩn để đánh giá tính hiệu quả của dữ liệu thử bằng cách cho thấy các dữ liệu thử có thể làm lộ ra tất cả các lỗi đơn giản có thể có của một chương trình, đó chính là tiêu chí *chất lượng đột biến*. Trong trường hợp bộ dữ liệu thử không thể xác định được tất cả các lỗi, tiêu chí chất lượng đột biến cung cấp một thước đo để xác định việc cải tiến bằng cách lựa chọn một bộ dữ liệu thử mới. Thước đo này cho phép kiểm soát, đánh giá và cải tiến dữ liệu thử dựa trên cơ sở *kiểm thử đột biến* [3]. Năm 1971, Dick Lipton đề xuất ra phương pháp kiểm thử đột biến, sau đó lĩnh vực này được đánh dấu sự ra đời và phổ biến bởi DeMillo [6].

Lý thuyết và kết quả thực nghiệm đã cho thấy rằng, kiểm thử đột biến là phương pháp hiệu quả để đánh giá chất lượng của các bộ dữ liệu thử. Tuy nhiên, kỹ thuật này có một số hạn chế. Một trong những hạn chế đó là chi phí của kiểm thử đột biến rất cao, tập trung ba bước tốn kém nhất là sản sinh đột biến, biên dịch các đột biến và kiểm thử từng phiên bản đột biến với các dữ liệu kiểm thử.

Các hệ thống kiểm thử đột biến truyền thống thường tạo ra số lượng lớn các chương trình đột biến. Điều này làm tăng chi phí thực thi do mỗi đột biến phải thực thi với ít nhất một ca kiểm thử.

Nhiều kỹ thuật được nghiên cứu nhằm giảm chi phí phân tích các đột biến. Các công nghệ và các kỹ thuật mới được đưa vào trong kiểm thử đột biến, nhằm hướng đến cải tiến từng công đoạn riêng lẻ để nâng cao tốc độ chung của kiểm thử đột biến. Các kỹ thuật được nghiên cứu theo ba chiến lược: *làm thông minh hơn, làm ít hơn và làm nhanh hơn*. Chiến lược làm ít hơn hướng đến lựa chọn những đột biến sao cho hiệu quả nhất nhưng vẫn đảm bảo chất lượng kiểm thử, gồm các kỹ thuật như đột biến lựa chọn [17], đột biến dựa trên ràng buộc (constrained mutation) [11], lấy mẫu đột biến (mutation sampling) [3]. Các kỹ thuật làm nhanh hơn hướng vào tự động hoá một số công đoạn và giảm tải ở các công đoạn chiếm nhiều chi phí tính toán, nhằm mục đích tạo ra và chạy các chương trình đột biến nhanh hơn các hệ thống chuẩn. Các phương pháp làm thông minh hơn mong muốn sử dụng các phương pháp "thông minh" để cải thiện khả năng thực hiện. Như kỹ thuật *đột biến yếu* [10], thay vì so sánh trạng thái của đột biến với chương trình gốc khi kết thúc chương trình thì đột biến yếu so sánh ngay sau câu lệnh đột biến. Hay sử dụng *kiến trúc phân tán* (distributed architectures), trong đó kiểm thử đột biến được áp dụng trên nhiều máy tính nhằm phân tán tính toán trên nhiều bộ vi xử lý. Krauser và các tác giả, khi áp dụng kiểm thử đột biến trên ngôn ngữ Fotran, đã đề xuất giải pháp thực thi tương tranh các đột biến trên các kiến trúc máy tính SIMD [18]. Fleyshgakker và Weiss giới thiệu thuật toán song song cho phép cải tiến rất hiệu quả việc thực thi các đột biến [20]. Offut và các đồng tác giả đề xuất giải pháp phân tán chi phí thực thi các đột biến trên kiến trúc máy tính MIMD [19]. Các kỹ thuật này được phát triển theo hướng phân tán việc thực thi đột biến trên nhiều máy tính.

Trong bài báo này, chúng tôi đề xuất giải pháp song song để cải thiện tốc độ thực thi đột biến khi áp dụng cho các mô hình Simulink. Điều này là có thể thực hiện được, bởi vì mỗi đột biến hay nhóm đột biến được thực hiện độc lập với các

đột biến khác và được thực thi riêng rẽ.

Bài báo được tổ chức gồm 5 mục: Mục 2 giới thiệu kiểm thử đột biến và ứng dụng kiểm thử đột biến cho Simulink; Mục 3 đề xuất giải pháp song song hóa việc thực thi đột biến trên máy tính đa nhân; Mục 4 trình bày một số kết quả thử nghiệm của giải pháp đề xuất và cuối cùng là Mục kết luận.

2. Kiểm thử đột biến cho các mô hình Simulink

2.1. Kiểm thử đột biến

Kiểm thử đột biến là một kỹ thuật hỗ trợ cho hoạt động kiểm thử, được tạo ra với mục đích kiểm tra, đánh giá bộ dữ liệu thử và giúp cho việc tạo ra các bộ dữ liệu thử có khả năng phát hiện lỗi của chương trình. Trong khi thực hiện kiểm thử đột biến, chúng ta tạo ra các phiên bản lỗi của chương trình gốc bằng cách chèn lỗi vào mã nguồn của chương trình cần kiểm thử. Mỗi phiên bản chỉ chứa đúng một lỗi và được gọi là một *đột biến* (mutant). Mỗi đột biến được tạo ra chỉ bởi một sự thay đổi cú pháp trong chương trình gốc, mỗi sự thay đổi cú pháp gọi là một luật hay còn được gọi là một *toán tử đột biến* (mutation operator). Các toán tử đột biến được định nghĩa sẵn để tạo ra sự thay đổi cú pháp dựa trên các lỗi mà các lập trình viên thường phạm phải. Ví dụ, thay một biến bởi một biến khác cùng kiểu, hoặc thay một toán tử số học bởi một toán tử số học khác, xoá toàn bộ các biểu thức; thay đổi câu lệnh… Các toán tử đột biến được xây dựng dựa trên ngôn ngữ dùng để cài đặt chương trình được kiểm thử.

Một cách cụ thể, đột biến P’ của chương trình gốc P là một chương trình tương tự với chương trình gốc P; P’ khác P do chỉ một thay đổi cú pháp trong chương trình. Chẳng hạn, trong chương trình gốc P có biểu thức (a>0 && b>0), P’ là đột biến của P bằng cách thay đổi cú pháp trong phép toán quan hệ, thay thế && bởi || ta có biểu thức trong P’ là (a>0 || b>0).

Khi tiến hành kiểm thử, lần lượt chương trình gốc P và đột biến P’ được thực hiện với bộ dữ liệu thử T. Nếu lỗi được chèn vào trong chương trình đột biến P’ bị nhận biết hoặc chương trình P và đột biến P’ cho ra kết quả khác nhau, đột biến P’ được gọi là bị diệt bởi bộ dữ liệu thử T và T được gọi là bộ dữ liệu thử thích hợp vì T có khả năng phát hiện được những khác nhau giữa chương trình gốc P (chương trình đúng) và đột biến P’ (chương trình sai). Nếu chương trình gốc P và đột biến P’ cho ra kết quả hoàn toàn giống nhau thì có hai khả năng xảy ra. Khả năng thứ nhất là bộ dữ liệu thử T không đủ tốt (T được gọi là bộ dữ liệu thử không thích hợp), chúng ta sẽ tiến hành thực hiện kiểm thử lại với một bộ dữ liệu thử tốt hơn. Khả năng thứ hai là chương trình P và đột biến P’ là những chương trình tương tự nhau, không có một bộ dữ liệu thử nào tìm ra được cách phân biệt sự khác nhau giữa chúng, khi đó đột biến P’ được gọi là *đột biến tương đương*. Trong cả hai trường hợp này, đột biến P’ được cho là *còn sống*.

Sau khi thực thi trên tất cả các bộ dữ liệu thử, tỷ lệ phần trăm của số các đột biến bị diệt chia cho số đột biến không tương đương được gọi là *tỷ lệ đột biến* (mutation score - MS). Mục tiêu của một kiểm thử viên là tạo ra bộ dữ liệu thử có MS bằng 1.0, nghĩa là tất cả các đột biến đều bị diệt. Khi đó, bộ dữ liệu thử phát hiện được tất cả các lỗi chèn vào trong chương trình, bộ dữ liệu như thế cũng có khả năng phát hiện các lỗi khác trong chương trình.

Kiểm thử đột biến được xây dựng dựa trên hai giả thuyết cơ bản: giả thuyết “lập trình viên giỏi” (competent programmer hypothesis - CPH) và giả thuyết “hiệu ứng liên kết” (coupling effect hypothesis - CEH) [14]. Giả thuyết lập trình viên giỏi cho rằng thông thường các lập trình viên đều rất giỏi và họ sẽ không bao giờ viết ra các chương trình một cách tuỳ tiện, cẩu thả. Trong quá trình viết lệnh cho chương trình, một lập trình viên nếu có sai sót, thì cũng sẽ tạo ra được chương trình rất gần với phiên bản đúng của chương trình, nghĩa là chỉ có một vài thay đổi về cú pháp rất nhỏ so với chương trình đúng, được gọi là các lỗi đơn giản. Do vậy, trong kiểm thử đột biến, chỉ các lỗi đơn giản được tạo ra bởi việc thực hiện các thay đổi nhỏ về cú pháp được áp dụng, nhằm biểu diễn các lỗi phạm phải bởi các “lập trình viên giỏi”. Giả thuyết hiệu ứng liên kết cho rằng các lỗi phức tạp được liên kết từ các lỗi đơn giản, như vậy bộ dữ liệu kiểm thử nếu đủ khả năng phát hiện tất cả các lỗi đơn giản trong chương trình thì cũng có khả năng phát hiện các lỗi phức tạp. Giả thuyết này đã được chứng minh bởi nhiều nghiên cứu lý thuyết và thực nghiệm [14], cho phép các đột biến sử dụng trong kiểm thử đột biến chỉ giới hạn các đột biến đơn giản, trong khi đó tạo ra được các kiểm thử hiệu quả xác định các lỗi phức tạp.

2.2. Kiểm thử đột biến cho các mô hình Simulink

Kể từ khi kiểm thử đột biến được đề xuất bởi DeMillo [6], kiểm thử đột biến cũng được áp dụng cho nhiều ngôn ngữ lập trình khác nhau. Ban đầu kiểm thử đột biến được đề xuất áp dụng cho các chương trình Fortran, sau đó được áp dụng cho các chương trình Ada [5], C [4], Java [12] và C# [8]. Kiểm thử đột biến cũng được áp dụng ở mức thiết kế, như đột biến đặc tả [7][9]. Trong bài báo [2], chúng tôi đã áp dụng kiểm thử đột biến cho các mô hình được thiết kế trong môi trường Simulink, trong đó chúng tôi đề xuất một bộ gồm 13 toán tử đột biến.

Các kết quả lý thuyết và thực nghiệm đã cho thấy rằng, kiểm thử đột biến là phương pháp hiệu quả để đánh giá chất lượng của các bộ dữ liệu thử. Tuy nhiên, kỹ thuật này có một số hạn chế. Một trong những hạn chế đó là chi phí của kiểm thử đột biến rất cao, tập trung ba bước tốn kém nhất là sản sinh đột biến, biên dịch các đột biến và kiểm thử từng phiên bản đột biến với các dữ liệu kiểm thử. Vì vậy, đã có nhiều giải pháp được nghiên cứu để cải thiện chi phí này. Một trong các giải pháp đó là tự động hóa các công đoạn kiểm thử đột biến. Do đó, để áp dụng kiểm thử đột biến cho các mô hình Simulink, chúng tôi thiết kế và cài đặt một framework nhằm tự động hóa các công đoạn sinh đột biến và sinh dữ liệu thử, gọi là MuSimulink [1].

Trong bất kỳ một hệ thống kiểm thử đột biến tự động nào cũng đều có một vài bước quan trọng mà kiểm thử viên cần tuân theo. Bởi vì, số đột biến sinh ra của mỗi chương trình rất lớn, vì vậy sẽ không thực tế để biên dịch và lưu trữ từng chương trình đột biến một cách riêng rẽ. Vì vậy hầu

hết các hệ thống đột biến, kể cả MuSimulink, đã được xây dựng như hệ thống thông dịch. Với phương pháp này, thay vì tạo, biên dịch và lưu trữ nhiều mô hình đột biến riêng lẻ, mô hình được dịch một lần vào một cấu trúc trung gian và mỗi đột biến được lưu trữ dưới dạng một mô tả ngắn về các thay đổi cần thiết để tạo ra các đột biến. Trong MuSimulink, những mô tả này được lưu trữ trong một cấu trúc dữ liệu gọi là bảng mô tả đột biến (MDT – Mutants Description Table), mỗi phần tử trong cấu trúc này là một bản ghi mô tả đột biến (MD). Các bước chính của qui trình kiểm thử đột biến được liệt kê dưới đây:

Tải mô hình: Một mô hình gốc P (cần kiểm thử) được tải lên để thử nghiệm, sẽ được phân tích để tạo một dạng trung gian cho sự thông dịch. Ở đây, chúng tôi sử dụng một đồ thị lưu trữ các phần tử của mô hình (khối, đường nối) và mối quan hệ giữa các khối.

Lựa chọn toán tử đột biến: Có thể một hoặc nhiều toán tử đột biến được lựa chọn để áp dụng cho mô hình P. Các kiểm thử viên thường sử dụng tất cả các toán tử đột biến có sẵn.

Sinh đột biến: Mỗi toán tử đột biến được chọn sẽ được áp dụng cho P và tạo một MDT mô tả tập hợp các đột biến M của P.

Sinh dữ liệu thử: Một tập hợp các ca kiểm thử T được nạp vào. Mỗi ca kiểm thử trong T có giá trị cho các biến đầu vào của P. T có thể được tạo ra thủ công hoặc tự động. MuSimulink cũng cung cấp chức năng sinh dữ liệu thử dựa vào các đặc tả cho các miền dữ liệu đầu vào.

Thực thi trên mô hình gốc: Mô hình gốc P được thực thi lần lượt trên mỗi ca kiểm thử và tạo ra một tập các đầu ra mong đợi O. Các kết quả dự kiến của P có thể được kiểm tra trong quá trình kiểm thử để xác định P có thực hiện đúng trên T hay không. Nếu có bất kỳ đầu ra không chính xác, thì P là không đúng và cần phải được sửa lại, quá trình kiểm tra lại được khởi động lại từ bước đầu tiên.

Thực thi mô hình đột biến: Mỗi mô hình đột biến được thông dịch và được thực thi trên mỗi ca kiểm thử trong T. Việc này tạo ra một tập đầu ra của đột biến O' gồm khoảng |M| x |T| phần tử. Nhưng vì số các đột biến đã bị diệt thường không được thực thi lại với các các ca kiểm thử mới nên O' thường nhỏ hơn |M| x |T|.

So sánh kết quả đầu ra: Mỗi phần tử của O' được so sánh với các phần tử của O được tạo ra từ cùng tập các dữ liệu thử. Nếu một đầu ra khác nhau, đột biến sẽ bị diệt. Sau khi so sánh đầu ra, nếu một số đột biến vẫn còn sống, thì hoặc là dữ liệu thử trong T là không thích hợp hoặc đột biến là tương đương với P, và không bao giờ có thể bị diệt. Xác định đột biến tương đương là một quy trình thủ công và tốn nhiều thời gian trong kiểm thử đột biến.

Phân tích kết quả: Kết quả kiểm thử được phân tích và nếu cần thiết, các ca kiểm thử bổ sung phải được thực hiện. Nếu tất cả các toán tử đột biến đã được áp dụng cho P và tất cả các đột biến đã bị diệt, thì không cần thiết phải tiếp tục kiểm thử. Nhưng nếu một hoặc nhiều đột biến không tương đương vẫn còn sống, ca kiểm thử bổ sung cần được thêm vào T và bước thích hợp của quá trình trên được lặp lại. Nếu như kết quả của kiểm thử, lỗi trong P được phát hiện ra, sau đó P phải được sửa đổi và quá trình kiểm thử lại được lặp lại. Các ca kiểm thử có thể được tái sử dụng để thử nghiệm bước tiếp theo.

Quá trình kiểm thử tiếp tục cho đến khi kiểm thử viên có một bộ dữ liệu thử với một tỷ lệ đột biến đạt yêu cầu hoặc bị buộc phải dừng lại do ràng buộc thời gian hoặc chi phí.

2.3. Giải pháp thực thi đột biến tuần tự

Để tiến hành thực thi đột biến, trong MuSimulink [2], chúng tôi đề xuất giải pháp thực thi đột biến theo cách tuần tự như Hình 1. Trong suốt quá trình thực thi đột biến, với mỗi ca kiểm thử các mô hình gốc phải được thực hiện trước và kiểm tra kết quả đầu ra. Nếu đầu ra không hợp lệ, tức mô hình gốc bị lỗi, sẽ dừng lại để sửa lỗi mô hình gốc. Nếu không, kết quả thực thi trên mô hình gốc được lưu lại vào một biến tạm. Sau đó, lần lượt với mỗi đột biến còn sống trong danh sách đột biến (MDT) sẽ được thực hiện với ca kiểm thử hiện tại. Nếu việc thực thi trên mô hình đột biến kết thúc không bình thường, hoặc báo lỗi thì xem như đột biến đó bị diệt và đánh dấu trạng thái bị diệt cho đột biến này. Nếu không, kết quả đầu ra của đột biến sẽ được so sánh với kết quả đầu ra của mô hình gốc. Trường hợp kết quả so sánh là khác nhau, đánh dấu trong danh sách trạng thái đột biến đã bị diệt, ngược lại giữ nguyên trạng thái còn sống của đột biến đó. Quá trình tiếp tục cho đến khi tất cả các đột biến bị diệt hoặc không còn ca kiểm thử nào.

```
Đầu vào:
Tệp chứa bộ dữ liệu thử T
Tập các toán tử đột biến được chọn thực thi O
Danh sách đột biến M
Đầu ra:
Danh sách đột biến còn sống/bị diệt
Thuật toán:
Đọc từ tệp chứa bộ dữ liệu thử T
foreach (testcase t in T)
    Thực thi mô hình gốc P trên t
    if (P lỗi) thông báo lỗi và kết thúc
    else ghi nhận kết quả Δ(P,t)
    endif
    foreach (toán tử op in O)
        foreach(đột biến còn sống m in M(op))
            Tạo mô hình đột biến P'
            Thực thi P' trên t
            Ghi nhận kết quả Δ(P', t)
            if (P' lỗi) đánh dấu m bị diệt
            elseIf (Δ(P', t) ≠ Δ(P, t))
                đánh dấu m bị diệt
            Else m còn sống.
            endif
        endfor
    endfor
endfor
Ghi kết quả thực thi vào tệp kết quả và hiển thị
```

Hình 1: *Thuật toán thực thi đột biến tuần tự*

Trong thuật toán ở Hình 1, chúng ta thấy rằng việc *"thực thi P' trên t"* và *"Ghi nhận kết quả* $\Delta(P', t)$*"* được thực hiện trên mỗi trường hợp thử. Bước này nếu được thực hiện tuần

tự, hết đột biến này đến đột biến khác, sẽ dẫn đến chi phí rất lớn. Vì vậy, đây là bước mà chúng tôi xem xét để đề xuất giải pháp xử lý song song.

3. Giải pháp song song cho MuSimulink

Chúng ta thấy rằng, hầu hết các chi phí tính toán của quá trình kiểm thử đột biến là thực thi mô hình gốc, thực thi mô hình đột biến, so sánh kết quả, và sinh dữ liệu thử. Vì việc *thực thi đột biến* được thực hiện mỗi một lần cho mỗi đột biến và mỗi ca kiểm thử, nó được xem như là một vòng lặp nội bộ. Vòng lặp nội bộ bao gồm việc thông dịch một đột biến với một ca kiểm thử, so sánh kết quả mô hình đột biến với kết quả mô hình gốc, và nếu chúng khác nhau, đột biến được xem là bị diệt. Cho đến nay, vòng lặp nội bộ này được xem là phần tính toán tốn kém nhất của kiểm thử đột biến, vì vậy đây chính là mục tiêu của giải pháp song song của chúng tôi.

Đầu vào:
Tệp chứa bộ dữ liệu thử `T`
Tập các toán tử đột biến được chọn thực thi `O`
Tệp chứa danh sách các đột biến D
Danh sách đột biến `M`
Đầu ra:
Danh sách đột biến còn sống/bị diệt
Thuật toán:
Đọc từ tệp chứa bộ dữ liệu thử `T`
Khởi động N worker W
foreach *(testcase t in T)*
  Thực thi mô hình gốc `P` *trên* `t`
  if *(P lỗi)*
    Thông báo lỗi, đóng worker và kết thúc
  else *ghi nhận kết quả* `Δ(P,t)`
endif
foreach *(toán tử* `op` ***in*** `O`*)*
  Gởi dữ liệu thử `t` *đến các worker*
  Gởi kết quả `Δ(P,t)` *đến các worker*
  foreach *(đột biến còn sống* `m` ***in*** `M(op)`*)*
    Gởi thông tin đột biến đến worker
    Tạo mô hình đột biến `P'`
    Thực thi `P'` *trên* `t`
    Ghi nhận kết quả `Δ(P', t)`
    if *(P' lỗi) đánh dấu* `m` *bị diệt*
    elseif `(Δ(P',t)≠Δ(P,t))`
      đánh dấu `m` *bị diệt*
    else `m` *còn sống.*
    endif
    Cập nhật số đột biến bị diệt
  endfor
  endfor
endfor
Đóng N worker
Ghi kết quả thực thi tệp kết quả và hiển thị

Hình 2: *Thuật toán thực thi đột biến song song*

Có hai cách để song song hóa việc thực thi đột biến là cung cấp cho mỗi bộ xử lý tất cả các ca kiểm thử và một tập hợp các đột biến, hoặc cung cấp mỗi bộ xử lý tất cả các đột biến và một tập hợp các ca kiểm thử. Ở đây, vì kích thước của tập dữ liệu thử T thường là tương đối nhỏ so với kích thước của tập các mô hình đột biến M, nên chúng tôi đã thực hiện phương pháp tiếp cận thứ nhất bằng cách song song trên các đột biến.

Chúng tôi sử dụng công cụ Parallel Computing Toolbox (PCT) của Matlab để cài đặt cho giải pháp song song được đề xuất. Công cụ PCT của Matlab cho phép nhiều *công nhân* (`worker`) làm việc trên một máy đơn có đa nhân. `Parfor` là cách đơn giản để thực hiện các công việc lặp đi lặp lại. Các lần lặp lại của `parfor` được thực hiện trên các `lab/worker`. Một `worker` là một thể hiện độc lập của MATLAB chạy trong một tiến trình hệ điều hành riêng biệt. Các `worker` được thực hiện trên các nhân bộ xử lý, và số lượng các `worker` không cần phải tương ứng với số lượng nhân.

Hình 2 là giải thuật được đề xuất nhằm song song hóa việc thực thi đột biến. Trong Hình 2, phần đóng khung đứt nét, việc thông dịch đột biến và so sánh kết quả giao cho các `worker` xử lý, mỗi `worker` chịu trách nghiệm một phần trong tổng số các đột biến. Việc thông dịch và so sánh kết quả thực thi đột biến nằm trong vòng lặp cho tất cả các đột biến còn sống.

Chiến lược song song được tổ chức gồm hai phần. Vòng lặp thực thi đột biến đặt trên các `worker`, phần thực thi mô hình gốc và phân tích kết quả đặt trên chương trình chính `client`. Giải thuật được đề xuất trong đó bộ vi xử lý chính (`client`) sẽ phân phối đột biến đến các `worker` và mỗi `worker` sẽ thực hiện việc thông dịch các đột biến trên mỗi ca kiểm thử.

Ban đầu, các `worker` chưa được sử dụng, bộ vi xử lý máy chính `client` tính toán và lưu trữ kết quả dự kiến từ mô hình gốc ban đầu. Sau đó, các máy chủ `client` bắt đầu gửi các thông tin khởi động đến các `worker`. Đối với mỗi ca kiểm thử, ca kiểm thử và kết quả ra dự kiến sẽ được gửi đến các `worker`, sau đó một khi một `worker` thông dịch đột biến trên ca kiểm thử, so sánh kết quả và trả về `client` trạng thái đột biến còn sống/bị diệt. Nếu tất cả các đột biến đã bị diệt hết, giải thuật kết thúc sớm và không cần gửi thêm ca kiểm thử nào đến `worker`, ngược lại, ca kiểm thử tiếp theo được gửi đi. Việc gửi các đột biến đến các `worker` nào được tự động thực hiện bởi `parfor` tùy theo tình trạng `worker` nào đang rảnh.

4. Thử nghiệm và đánh giá

Chúng tôi tiến hành cài đặt giải pháp đề xuất và thử nghiệm cho 7 mô hình Simulink trên hai máy tính đa nhân, kết quả thử nghiệm được trình bày trong Bảng 1 và Bảng 2.

Trong Bảng 1, chúng tôi tiến hành thử nghiệm trên máy tính A với cấu hình 2 nhân, 4 GB RAM, Intel ® Core ™ i5-3337U CPU @ 1.80GHz. Trong Bảng 2, chúng tôi tiến hành thử nghiệm trên máy tính B với cấu hình 4 nhân, 2 GB RAM, Intel ® Core ™2 Quad CPU Q6600 @ 2.40GHz.

Nhìn vào kết quả thu được từ việc thực thi các mô hình ở hai chế độ tuần tự và song song với cùng bộ dữ liệu thử như nhau, chúng ta thấy hiệu quả của giải pháp song song đã đề xuất. Thời gian thực thi đột biến khi áp dụng giải pháp song song giảm từ 57% đến 73% trên máy A, và giảm từ 58% đến 81% trên máy B.

Kết quả cho thấy tốc độ thực thi đột biến được cải thiện khi thực thi song song, nhưng kết quả này không phải thay đổi tuyến tính theo số nhân có trên máy, mà còn phụ thuộc vào loại CPU hỗ trợ.

Bảng 1: *Thời gian thực thi đột biến trên máy A*

Tên mô hình	Số đột biến sinh ra	Thời gian (s)		Hiệu quả
		Tuần tự	*Song song*	
Constant_Accel	162	12.980	5.57	57%
CheckInputs	245	35.334	12.66	64%
Motor_Model	178	12.324	3.98	68%
Quadratic	225	118.592	31.79	73%
sldemo_f14	584	152.179	57.43	62%
SmokeDetector	484	115.239	33.49	71%
Tiny	213	51.496	14.45	72%

Bảng 2: *Thời gian thực thi đột biến trên máy B*

Tên mô hình	Số đột biến sinh ra	Thời gian (s)		Hiệu quả
		Tuần tự	*Song song*	
Constant_Accel	162	17.509	7.27	58%
CheckInputs	245	43.67	12.08	72%
Motor_Model	178	14.574	5.47	62%
Quadratic	225	143.433	28.13	80%
sldemo_f14	584	241.943	47.38	80%
SmokeDetector	484	132.55	25.78	81%
Tiny	213	63.826	15.02	76%

5. Kết luận và hướng phát triển

Kiểm thử đột biến là một phương pháp hiệu quả trong việc đánh giá các bộ dữ liệu thử. Tuy nhiên, một trong các hạn chế của phương pháp này là chi phí cho việc thực thi với số lượng đột biến lớn trên tất cả các ca kiểm thử. Để khắc phục hạn chế này cho công cụ MuSimulink, chúng tôi đã đề xuất giải pháp thực thi song song áp dụng trên máy tính đa nhân. Sử dụng công cụ PCT của Matlab nên các công việc được phân phối đến các `worker` làm việc do bản thân công cụ tự điều phối trên cơ sở cân bằng tải giữa các `worker`. Kết quả thử nghiệm trên một số mô hình đã cho thấy giải pháp đề xuất khá hiệu quả. Công việc tiếp theo của chúng tôi sẽ tiến hành cài đặt giải pháp song song trên nhiều máy tính, và nghiên cứu để áp dụng các chiến lược phân chia công việc đến các `worker`.

Tài liệu tham khảo

[1] Lê Thị Mỹ Hạnh, Nguyễn Thanh Bình, Tự động sinh đột biến cho các mô hình Simulink/Matlab, *Kỷ yếu Hội thảo Quốc gia về CNTT&TT lần thứ 16* 11/2013 (đang xuất bản).

[2] Le Thi My Hanh, Nguyen Thanh Binh, Mutation Operators for Simulink models *Proceedings of the Fourth International Conference on Knowledge and Systems Engineering (KSE 2012)*, No: 4, Pages: 54-60, 2012.

[3] T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation Analysis," Georgia Institute of Technology, Atlanta, Georgia, *Technique Report GIT-ICS-79/08,* 1979.

[4] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of Mutant Operators for the C Programming Language," Purdue University, West Lafayette, Indiana, *Technique Report SERC-TR-41-P,* March 1989.

[5] J. H. Bowser, "Reference Manual for Ada Mutant Operators," Georiga Institute of Technology, Atlanta, Georgia, *Technique Report GITSERC-88/02,* 1988.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer", *IEEE Computer,* 11(4), April 1978.

[7] M. E. Delamaro, "Proteum - A Mutation Analysis Based Testing Environment", Master Thesis, University of S~ao Paulo, Sao Paulo, Brazil, 1993.

[8] Derezinska, "Quality Assessment of Mutation Operators Dedicated for C# Programs," *in Proceedings of the 6th International Conference on Quality Software (QSIC'06),* Beijing, China, 27-28 October 2006.

[9] A. S. Gopal and T. A. Budd, "Program Testing by Speci?cation Mutation", University of Arizona, Tucson, Arizona, *Technical Report TR83-17,* 1993.

[10] W. E. Howden, "Weak Mutation Testing and Completeness of TestSets", *", IEEE Transactions on Software Engineering,* vol. 8, no. 4, pp. 371–379, July 1982.

[11] S. Hussain, "Mutation Clustering, *Masters Thesis,* King's College London, Strand, London, 2008.

[12] S. Kim, J. A. Clark, and J. A. McDermid, "The Rigorous Generation of Java Mutation Operators Using HAZOP," in *Proceedings of the 12th International Cofference Software and Systems Engineering and their Applications (ICSSEA 99),* Paris, France, 29 November-1 December 1999.

[13] P. Mathur, "Performance, Effectiveness, and Reliability Issues in Software Testing," *in Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC'79),* Tokyo, Japan, 11-13 September 1991, pp. 604–605.

[14] A. J. Offutt, "The Coupling Effect: Fact or Fiction," *ACM SIGSOFT Software Engineering Notes,* vol. 14, no. 8, pp. 131–140, December1989.

[15] A. J. Offutt and K. N. King, "A Fortran 77 Interpreter for Mutation Analysis," *ACM SIGPLAN Notices,* vol. 22, no. 7, pp. 177–188, July 1987.

[16] J. Offutt, G. Rothermel, and C. Zapf, "An Experimental Evaluation of Selective Mutation," *in Proceedings of the 15th International Conference on Software Engineering (ICSE'93). Baltimore, Maryland: IEEE Computer Society Press,* May 1993, pp. 100–107.

[17] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation Analysis Using Mutant Schemata," *in Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93),* Cambridge, Massachusetts, 1993, pp. 139–148.

[18] E. W. Krauser, A. P. Mathur, and V. J. Rego, "High Performance Software Testing on SIMD Machines," *IEEE Transactions on Software Engineering,* vol. 17, no. 5, pp. 403–423, May 1991.

[19] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation Testing of Software Using a MIMD Computer," *in Proceedings of the International Conference on Parallel Processing,* Chicago, Illinois,August 1992, pp. 255–266.

[20] S. N. Weiss and V. N. Fleyshgakker, "Improved Serial Algorithms for Mutation Analysis," *ACM SIGSOFT Software Engineering Notes,* vol. 18, no. 3, pp. 149–158, July 1993.

(BBT nhận bài: 25/12/2013, phản biện xong: 05/01/2014)