PHÂN TÍCH VÀ ĐÁNH GIÁ HIỆU NĂNG HOẠT ĐỘNG CỦA CAPE

Hà Viết Hải[1], Trần Văn Long[2]
[1]Trường Đại học Sư phạm, Đại học Huế, Việt Nam
[2]Trường Cao đẳng Công nghiệp Huế, Việt Nam
haviethai@gmail.com, tvlong@hueic.edu.vn

TÓM TẮT— *MPI (Message Passing Interface) và OpenMP là hai công cụ được sử dụng rộng rãi để phát triển các chương trình song song. Các chương trình MPI có ưu điểm là có hiệu năng cao nhưng khó phát triển trong khi OpenMP rất mạnh và dễ sử dụng. Tuy nhiên nhược điểm lớn nhất của OpenMP là chỉ có thể chạy trên các kiến trúc sử dụng bộ nhớ chia sẻ. CAPE (Checkpointing Aided Parallel Execution) là một hướng tiếp cận sử dụng kỹ thuật đánh dấu tiến trình (Checkpointing) để cài đặt OpenMP trên các kiến trúc sử dụng bộ nhớ phân tán. Bài báo này đi sâu vào việc phân tích và đánh giá hiệu năng hoạt năng hoạt động của CAPE thông qua việc so sánh mô hình hoạt động của nó với mô hình của MPI, từ đó đưa ra những nhận định cho việc sử dụng CAPE một cách hiệu quả.*

Từ khóa— *CAPE, OpenMP, MPI, parallel computing, high performance computing, lập trình song song, tính toán hiệu năng cao.*

I. GIỚI THIỆU

Để giảm thiểu khó khăn cho các lập trình viên khi phát triển các ứng dụng song song, các công cụ lập trình song song ở mức trừu tượng cao và dễ sử là một đòi hỏi bức thiết. MPI (Message Passing Interface) [1] và OpenMP [2] là hai công cụ trình được sử dụng rộng rãi để đáp ứng yêu cầu này. Các chương trình MPI có hiệu năng cao nhưng khó phát triển trong khi OpenMP rất mạnh và dễ sử dụng. Tuy nhiên nhược điểm lớn nhất của OpenMP là chỉ có thể chạy trên các kiến trúc sử dụng bộ nhớ chia sẻ.

Đã có nhiều nỗ lực để cài đặt OpenMP trên các kiến trúc sử dụng bộ nhớ phân tán nhưng chưa có phương án nào thành công trên cả hai mặt là tương thích hoàn toàn với chuẩn OpenMP và có hiệu năng cao. Những phương án nổi bật nhất có thể kể đến là sử dụng SSI [3]; SCASH [4]; sử dụng mô hình RC [5]; biên dịch thành MPI [6][7]; sử dụng Global Array [8]; Cluster OpenMP [9]. Phương án sử dụng một ảnh chung hệ thống (Single System Image - SSI) như một bộ nhớ chung cho các tiến trình trong hệ thống là hướng trực quan nhất để cài đặt OpenMP trên các kiến trúc phân tán. Do có bộ nhớ chung, các chương trình OpenMP có thể dễ dàng được biên dịch để chạy trên các tiến trình trên các máy tính khác nhau của hệ thống. Tuy nhiên, do việc truy cập bộ nhớ chung trên mạng và việc đồng bộ hóa các truy cập này tiêu tốn nhiều thời gian, tiếp cận này không thể cung cấp hiệu năng cao. Một thử nghiệm thực tế [10] đã cho thấy tốc độ xử lý của chương trình lại tỷ lệ nghịch với số tiến trình, thay vì tỷ lệ luận như mong đợi. Để giảm sự trả giá về thời gian do đặt toàn bộ không gia nhớ của các tiến trình lên trên một SSI, các phương án như SCASH chỉ ánh xạ các biến chia sẻ giữa các tiến trình lên phần bộ nhớ chung, còn tiến trình vẫn sử dụng bộ nhớ cục bộ. Một phương án khác theo hướng này là sử dụng mô hình bộ nhớ đồng bộ hóa trễ (Relaxed Consistency Memory Model). Tuy nhiên, các phương án này đều gặp khó khăn trong việc tự động xác định các biến chia sẻ cũng như trong việc cài đặt một số cấu trúc bộ nhớ chia sẻ khác nên chúng đều không thể tương thích hoàn toàn với chuẩn OpenMP. Các phương án biên dịch OpenMP thành dạng MPI có ưu điểm về hiệu năng nhưng vẫn không thể cài đặt được tất cả các cấu trúc và chỉ thị của OpenMP. Ngay cả với Cluster OpenMP, một sản phẩm thương mãi của Intel cũng đòi hỏi phải sử dụng thêm các chỉ thị riêng của nó (không nằm trong chuẩn OpenMP) trong một số trường hợp và vì vậy, nó cũng chưa cung cấp được một cài đặt tương thích hoàn toàn với OpenMP.

CAPE (Checkpointing Aide Parallel Execution) là một hướng tiếp cận khác để cài đặt OpenMP trên các kiến trúc sử dụng bộ nhớ phân tán. CAPE sử dụng kỹ thuật đánh dấu tiến trình (checkpointing) để phân phối công việc trong các khối chương trình song song đến các tiến trình khác nhau của hệ thống và để xử lý việc trao đổi dữ liệu chia sẻ một cách tự động. Hai phiên bản của CAPE [11][12] đã được phát triển và thử nghiệm đã chứng minh tính khả thi cũng như hiệu năng cao của CAPE.

Bài báo này đi sâu vào việc phân tích và đánh giá hiệu năng hoạt năng hoạt động của CAPE thông qua việc so sánh mô hình hoạt động của nó với mô hình của MPI, từ đó đưa ra những nhận định cho việc sử dụng CAPE một cách hiệu quả. Lý do của việc so sánh và đánh giá này là khá rõ ràng: so sánh với hiệu năng của MPI tức là so sánh với giải pháp có hiệu năng cao nhất trên các kiến trúc sử dụng bộ nhớ phân tán.

Phần còn lại của bài báo được tổ chức như sau: chương 2 trình bày một số vấn đề liên quan đến vấn đề nghiên cứu, bao gồm các giới thiệu sơ lược về MPI, OpenMP và kỹ thuật đánh dấu tiến trình. Chương tiếp theo là phần giới thiệu về CAPE dựa trên kỹ thuật đánh dấu tiến trình gia tăng rời rạc. Chương 4 là một so sánh lý thuyết về mô hình hoạt động của CAPE và MPI. Chương 5 nêu một kết quả thực nghiệm để kiểm chứng cho các phân tích ở chương 4. Chương cuối cùng nêu kết luận và một số định hướng nghiên cứu trong tương lai.

II. CÁC VẤN ĐỀ LIÊN QUAN

2.1 MPI

MPI [1] là một giao diện lập trình (API) được phát triển từ khá lâu. MPI cung cấp một thư viện căn bản nhất cho việc tổ chức một chương trình song song theo kiểu đa tiến trình. Chúng bao gồm các câu lệnh để khởi tạo môi trường MPI, phân chia các xử lý vào các đoạn mã song song, mỗi phần chạy trên một tiến trình; các câu lệnh để trao đổi dữ liệu giữa các tiến trình; các câu lệnh đồng bộ hóa các tiến trình... Mô hình xử lý thường được dùng là một tiến trình chính (master) và nhiều tiến trình phụ (slave). Thông thường, tiến trình chính tại máy (hoặc CPU) nguyên thủy ban đầu, làm nhiệm vụ khởi tạo chương trình, phân chia công việc và nhận kết quả tính toán từ các tiến trình phụ. Các tiến trình phụ chạy ở các máy (hoặc CPU) phụ, mỗi tiến trình nhận dữ liệu ban đầu tiến trình chính, thực hiện việc tính toán được giao và gửi kết quả về tiến trình chính.

Tuy có khả năng cung cấp hiệu năng cao và có thể chạy trên cả các kiến trúc sử dụng bộ nhớ chia sẻ lẫn phân tán, nhưng các chương trình MPI đòi hỏi người lập trình phải tự phân chia chương trình thành các khối cho tiến trình chính và phụ bằng các câu lệnh tường minh. Các công việc khác như gửi và nhận dữ liệu, đồng bộ hóa tiến trình… cũng cần được chỉ rõ bằng các câu lệnh và tham số cụ thể. Điều này làm khó khăn cho người lập trình ở hai mặt. Thứ nhất là sự đòi hỏi phải tổ chức ngay từ đầu chương trình thành theo cấu trúc song song, một việc phức tạp hơn nhiều so với việc tổ chức chương trình tuần tự vốn quen thuộc với tư duy và khả năng của đa số lập trình viên. Điều này cũng gây khó khăn cho việc song song hóa các chương trình tuần tự có sẵn, vì nó phá vỡ nghiêm trọng cấu trúc ban đầu của chương trình. Điều khó khăn thứ hai chính là sự rắc rối của các câu lệnh MPI, với một loạt các tham số khá khó hiểu cho mỗi câu lệnh. Do các khó khăn này, tuy có hiệu năng cao và đã cung cấp một cách thức trừu tượng hóa hơn nhưng MPI vẫn mới được xem là *assembler* của việc lập trình song song.

2.2 OpenMP

OpenMP [2] là một API cung cấp một mức trừu tượng hóa cao hơn nhiều so với MPI để phát triển các chương trình song song. Nó bao gồm một tập các biến môi trường, các chỉ thị và hàm, được xây dựng để hỗ trợ việc dễ dàng biến một chương trình tuần tự trên ngôn ngữ cơ sở là C/C++ hoặc Fortran thành một chương trình song song. Người lập trình có thể bắt đầu viết chương trình trên ngôn ngữ cơ sở, biên dịch và chạy thử nó. Sau đó, việc song song hóa có thể được tiến hành từng bước bằng cách thêm các chỉ thị biên dịch OpenMP vào những nơi cần thiết. Các chỉ thị này sẽ được một trình biên dịch C/C++ hoặc Fortran có hỗ trợ OpenMP biên dịch thành dạng mã thực thi. OpenMP cung cấp các chỉ thị để song song hóa những đoạn mã tiềm năng nhất cho việc xử lý song song, bao gồm các vòng lặp `for`, các đoạn mã được thực hiện song song như nhau và không như nhau trên các đơn vị xử lý.

OpenMP sử dụng mô hình thực hiện `fork-join` với cấu trúc song song cơ sở là thread. Ban đầu, chương trình chỉ có một thread chính, thực hiện các đoạn mã tuần tự. Mỗi khi gặp một chỉ thị song song của OpenMP, thread chính sinh ra một đội (team) làm việc bao gồm chính nó (thread chính) và một tập các thread phụ (pha `fork`) và công việc được phân chia cho mỗi thread trong đội. Sau khi các thread phụ hoàn thành phần mã của nó, kết quả được cập nhật vào không gian nhớ của thread chính và chúng có thể kết thúc hoạt động (pha `join`). Như vậy, sau pha này thì chương trình chỉ còn lại một tiến trình như ban đầu. Các pha `fork-join` có thể được tiến hành nhiều lần trong một chương trình và có thể lồng nhau.

Do sử dụng cấu trúc cơ sở là thread, mặc nhiên mô hình bộ nhớ của OpenMP là bộ nhớ chia sẻ, trong đó không gian nhớ được sử dụng chung giữa các tiến trình. Tuy nhiên, để tăng tốc độ tính toán, OpenMP sử dụng mô hình bộ nhớ đồng bộ trễ (Relaxed Consistency). Các tiến trình có thể sử dụng bộ nhớ cục bộ riêng để tăng tốc độ truy cập. Việc đồng bộ hóa giữa không gian nhớ của các tiến trình được thực hiện ngầm ở đầu và cuối các cấu trúc song song, hoặc tường minh bằng cách sử dụng chỉ thị `flush`. Tuy nhiên, hiện tại thì OpenMP vẫn chỉ mới được cài đặt một cách hoàn chỉnh dưới dạng sử dụng bộ nhớ chia sẻ (multiCPU, multicore) do sự phức tạp của việc cài đặt tất cả các yêu cầu của OpenMP trên các kiến trúc sử dụng bộ nhớ khác. Đây chính là động lực để nhiều nghiên cứu với mục tiêu cài đặt OpenMP trên các kiến trúc phân tán như đã được đề cập trong phần giới thiệu.

2.3 Kỹ thuật đánh dấu tiến trình

Đánh dấu tiến trình (chụp ảnh tiến trình) là kỹ thuật chụp và lưu trữ trạng thái của một chương trình đang vận hành sao cho nó có khả năng khôi phục lại trạng thái ở các thời điểm sau đó [13]. Kỹ thuật đánh dấu tiến trình được sử theo nhiều hướng khác nhau như gia tăng khả năng chịu lỗi của chương trình theo mô hình `rollback`, di trú tiến trình, backup hệ thống…

Kỹ thuật đánh dấu tiến trình có thể được phân chia vào hai nhóm: đánh dấu đầy đủ (Complete Checkpointing) và đánh dấu gia tăng (Incremental Checkpointing). Ở nhóm thứ nhất, toàn bộ không gian nhớ của tiến trình và một số thông số hệ thống khác của nó được chụp và lưu trữ cho mỗi ảnh của tiến trình (checkpoint). Đây là kiểu đơn giản nhưng các ảnh có kích thước lớn và dễ có dư thừa dữ liệu cho những ảnh chụp liên tiếp nhau của tiến trình. Ở nhóm thứ hai, mỗi ảnh của tiến trình chỉ lưu lại những phần không gian nhớ và thông số hệ thống của tiến trình đã bị cập nhật kể từ khi khởi tạo chương trình hoặc kể từ lần chụp ảnh trước. Để biết được những phần bị cập nhật này, chương trình được chụp ảnh thường được giám sát và chụp ảnh bởi một tiến trình chụp ảnh (checkpointer) chạy song song với nó. Trong kỹ thuật này, các ảnh tiến trình phải được chụp một cách kế tiếp nhau.

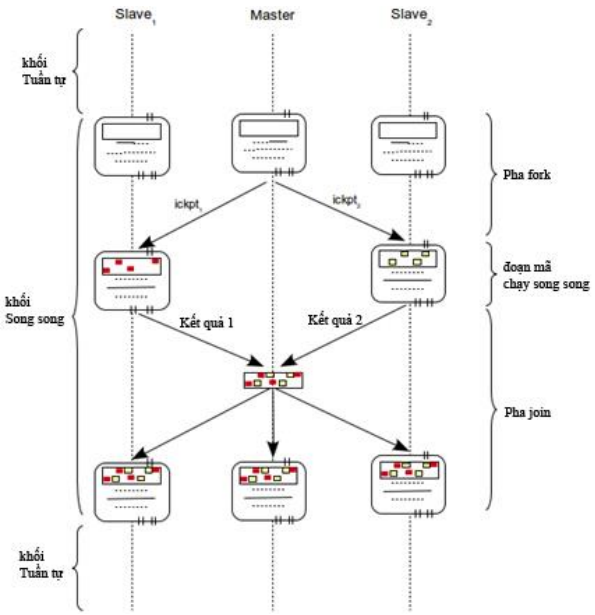
Để phục vụ một cách tối ưu cho CAPE, trong đó có yêu cầu chụp ảnh từng phần không liên tục của tiến trình, chúng tôi đã phát triển kỹ thuật Đánh dấu tiến trình Gia tăng rời rạc (Discontinuos Incremental Checkpointing—

DICKPT) [14]. Kỹ thuật này đặt cơ sở trên kỹ thuật đánh dấu tiến trình gia tăng nói trên, với bổ sung khả năng chụp ảnh từng đoạn rời rạc của một tiến trình. Bên cạnh khả năng mới này, các phân tích và số liệu thực nghiệm đã chứng minh hiệu suất cao của nó trong cả hai mặt là giảm kích thước ảnh chụp và giảm thời gian thực hiện quá trình chụp ảnh.

III. CAPE DỰA TRÊN KỸ THUẬT ĐÁNH DẤU TIẾN TRÌNH GIA TĂNG RỜI RẠC

3.1 Mô hình vận hành

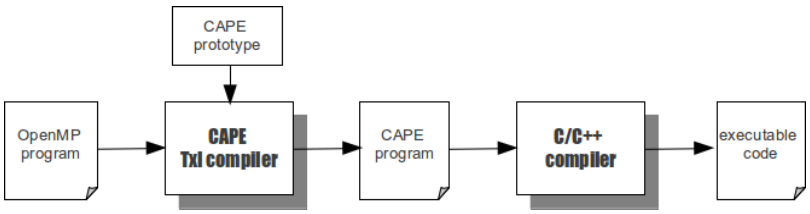
CAPE là một hướng mới để cài đặt OpenMP trên các hệ thống phân tán. CAPE sử dụng đơn vị song song cơ sở là tiến trình, thay cho thread như trong OpenMP nguyên bản để có thể vận hành trên các hệ thống sử dụng bộ nhớ phân tán. Với CAPE, tất cả các nhiệm vụ quan trọng nhất của quá trình mô hình `fork-join` đều được cài đặt một cách tự động dựa trên kỹ thuật đánh dấu tiến trình, bao gồm việc phân chia công việc cho các tiến trình, trích rút kết quả thực hiện trên các tiến trình phụ và cập nhật kết quả vào không gian nhớ của tiến trình chính… Phiên bản đầu tiên của CAPE sử dụng các ảnh chụp tiến trình đầy đủ đã chứng minh được khả năng thực hiện của CAPE. Tuy nhiên, do các ảnh chụp tiến trình đầy đủ có kích thước lớn dẫn đến lưu lượng lớn dữ liệu được chuyển giữa các tiến trình. Mặt khác, việc trích rút kết quả tính toán tại các tiến trình phụ dựa trên sự so sánh các ảnh chụp tiến trình đầy đủ có số lượng phép so sánh rất lớn. Cả hai điều này đã làm giảm hiệu suất vận hành cũng như khả năng mở rộng (scalability) của CAPE. Những yếu điểm này đã được khắc phục trong phiên bản thứ hai của CAPE, dựa trên kỹ thuật DICKPT. Hình 1 mô tả mô hình hoạt động của phiên bản này, trên đó hệ thống bao gồm 3 node, một node chạy tiến trình chính (Master) và hai node phụ, mỗi node chạy một tiến trình phụ (Slave) thực hiện công việc tính toán. Để đơn giản hóa cho việc trình bày, trong các khối mã song song, tiến trình chính chỉ làm nhiệm vụ phân chia công việc tới các tiến trình phụ và nhận kết quả từ chúng. Điều này là không bắt buộc và tiến trình chính có thể thực hiện một phần việc tính toán trong các khối mã song song.



Hình 1. Mô hình hoạt động của CAPE sử dụng kỹ thuật đánh dấu tiến trình gia tăng rời rạc

Khởi đầu, chương trình được khởi tạo trên tất cả các node và các đoạn mã tuần tự được thực hiện như nhau trên tất cả các node đó. Khi gặp một cấu trúc song song OpenMP, tiến trình chính phân chia công việc tới các tiến trình phụ bằng cách gửi đến mỗi tiến trình phụ một ảnh chụp tiến trình gia tăng. Lưu ý là ảnh chụp này có kích thước rất bé, thường là chỉ vài byte, do nó được lấy chứa kết quả của việc thực hiện một số câu lệnh đơn giản và không làm thay đổi nhiều không gian nhớ. Ở mỗi node phân tán, tiến trình phụ nhận ảnh chụp tiến trình, tích hợp nó vào trong không gian nhớ của mình và khởi tạo quá trình đánh dấu tiến trình DICKPT. Sau đó tiến trình phụ thực hiện việc tính toán được giao và trích rút kết quả thực hiện bằng một ảnh chụp tiến trình. Kết quả này được gửi về tiến trình chính, nơi nhận và tổ hợp chúng lại, sau đó tích hợp vào trong không gian nhớ của nó, cũng như gửi lại cho các tiến trình phụ để đồng bộ hóa không gian nhớ của tất cả các tiến trình, chuẩn bị cho việc thực hiện các đoạn mã tiếp theo của chương trình. Sau bước này, không gian nhớ của mỗi tiến trình đều đã có kết quả thực hiện của phần mã song song, vì thế, chúng xem như đều đã thực hiện xong phần việc của đoạn mã này.

3.2 Mô hình chuyển đổi cho cấu trúc parallel for



Hình 2. Quá trình biên dịch chương trình OpenMP thành mã thực thi trên nền CAPE

Các mô hình chuyển đổi được trình biên dịch của CAPE sử dụng để chuyển đổi các cấu trúc OpenMP sang dạng mã tương đương sử dụng các hàm CAPE, trong đó không còn các chỉ thị OpenMP nữa. Kết quả nhận được sau đó lại tiếp tục được biên dịch bởi một trình biên dịch C/C++ bình thường thành dạng mã thực thi và có thể chạy trên nền hỗ trợ CAPE. Hình 2 biểu diễn các bước trong quá trình biên dịch một chương trình OpenMP (với ngôn ngữ cơ sở là C/C++) thành mã thực thi trên nền CAPE. Các cấu trúc trong phiên bản CAPE hiện tại được xây dựng với giả thiết là

các câu lệnh song song thỏa mãn các điều kiện Bernstein, tức là mỗi phần mã thành phần có thể được thực hiện một cách độc lập, đầu vào và đầu ra của chúng không liên quan nhau và trong quá trình thực hiện, không có sự trao đổi dữ liệu với nhau.

Trong số các cấu trúc song song OpenMP, cấu trúc `parallel for` là cấu trúc song song quan trọng nhất vì hai lý do: 1) đây là cấu trúc song song được sử dụng nhiều nhất và 2) nó có thể đóng vai trò là cấu trúc trung gian để chuyển đổi các cấu trúc song song OpenMP khác bao gồm `parallel, sections`. Có nghĩa là thay vì xây dựng các mô hình để chuyển đổi trực tiếp các cấu trúc này, chúng trước hết được chuyển sang dạng cấu trúc `parallel for`. Với CAPE, một cấu trúc `parallel for` được tự động biên dịch thành một tập các câu lệnh, trong đó sử dụng các hàm cơ sở của CAPE, mô hình của việc chuyển đổi được trình bày trong hình 3.Để đơn giản cho việc trình bày, mô hình được xây dựng với giả thiết là số bước lặp của câu lệnh `for(A; B; C)` bằng với số tiến trình phụ. Dưới đây là liệt kê tên và công dụng của các hàm cơ sở CAPE được sử dụng trong mô hình.

1. `master()`: trả về `true` nếu mã đang được chạy trên tiến trình chính, `false` nếu mã đang chạy trên tiến trình phụ.

2. `start()`: xóa bộ đệm của checkpointer, khởi tạo một checkpoint mới. Đây là hàm được cung cấp bởi checkpointer DICKPT.

3. `create(ckpt)`: tạo một checkpoint và lưu vào `ckpt`. Đây là hàm được cung cấp bởi checkpointer DICKPT.

4. `send(ckpt, node)`: gửi checkpoint `ckpt` đến `node`.

5. `stop()`: dừng quá trình lấy checkpoint, xóa bộ đệm của checkpointer. Đây là hàm được cung cấp bởi checkpointer DICKPT.

6. `wait_for(ckpt)`: chờ và nhận các checkpoint con của checkpoint ckpt từ các tiến trình phụ, gộp chúng lại thành `ckpt`. Hàm này chỉ được gọi trên tiến trình chính.

7. `inject(ckpt)`: cập nhật các giá trị lưu trong checkpoint ckpt vào trong không gian nhớ của tiến trình hiện thời.

8. `last_parallel()`: trả về true nếu cấu trúc hiện tại là cấu trúc song song cuối cùng trong chương trình và trả về false trong trường hợp ngược lại.

9. `merge(ckpt1, ckpt2)`: hợp checkpoint $ckpt_2$ vào checkpoint $ckpt_1$.

10. `broadcast(ckpt)`: gửi checkpoint `ckpt` đến tất cả các tiến trình phụ. Hàm này chỉ được gọi từ tiến trình chính.

11. `receive(ckpt)`: nhận checkpoint `ckpt`.

```
# pragma omp parallel for
    for ( A ; B ; C )
        D
```

↓ Biên dịch tự động ↓

```
1   if ( master ( ) )
2       start ( )
3       for ( A ; B ; C )
4           create ( before )
5           send ( before, slave_x )
6       create ( final )
7       stop ( )
8       wait_for ( after )
9       inject ( after )
10      if ( ! last_parallel ( ) )
11          merge ( final, after )
12          broadcast ( final )
13  else
14      receive ( before )
15      inject ( before )
16      start ( )
17      D
18      create ( after_i )
19      stop ( )
20      send ( after_i, master )
21      if ( ! last_parallel ( ) )
22          receive ( final )
23          inject ( final )
24      else
25          exit
```

Hình 3. Mô hình chuyển đổi cho cấu trúc `parallel for`

Các bước chính của đoạn mã CAPE được giải thích dưới đây.

- Ở tiến trình chính (câu lệnh 1—12): phần đầu của vòng lặp `for` được thực hiện ở trên tiến trình này. Tại mỗi bước lặp, một checkpoint gia tăng rời rạc được lấy và gửi đến một tiến trình phụ. Sau đó tiến trình chính đợi và lấy các kết quả thực hiện thân của vòng lặp forđược gửi về từ các tiến trình phụ. Các kết quả này được gộp lại, cập nhật vào không gian nhớ của tiến trình chính, đồng thời được gửi đến tất cả các tiến phụ nếu cần thiết. Như vậy, sau bước này, không gian nhớ của các tiến trình đều đã chứa kết quả thực hiện của vòng lặp for.
- Ở tiến trình phụ (câu lệnh 13—25): tại đây, phần đầu của câu lệnh `for` không được thực hiện. Thay vào đó, mỗi tiến trình nhận một checkpoint gia tăng rời rạc từ tiến trình chính và cập nhật và không gian nhớ của nó. Như vậy, sau bước này, tiến trình iđã ở trạng thái sau khi thực hiện phần đầu của vòng lặp `for` được `i` bước. Tiếp theo, tiến trình khởi tạo một checkpoint gia tăng rời rạc và thực hiện thân của vòng lặp `for` (câu lệnh 16—17). Sau khi thực hiện xong việc tính toán, một checkpoint được tạo ra để trích rút các kết quả thực hiện được bởi thân của vòng lặp. Checkpoint này được gửi về tiến trình chính. Nếu đây chưa phải là cấu trúc song song cuối cùng của chương trình thì tiến trình phụ nhận checkpoint lưu giữ kết quả thực hiện của toàn bộ vòng lặp `for` (trên tất cả các node), cập nhật vào không gian nhớ của mình để chuẩn bị cho các câu lệnh tiếp theo.

IV. SO SÁNH MÔ HÌNH HOẠT ĐỘNG GIỮA MPI VÀ CAPE

4.1 Sự khác nhau

Điểm khác biệt lớn nhất giữa CAPE và MPI là cách thức xác định dữ liệu cần trao đổi và cách thức cập nhật dữ liệu này vào trong không gian nhớ của một tiến trình. Trong khi MPI yêu cầu người dùng xác định một cách tường

minh các dữ liệu này (qua địa chỉ, độ dài, kiểu dữ liệu) thì CAPE thực hiện tất cả một cách tự động thông qua các ảnh chụp tiến trình. Với CAPE, kết quả thực hiện của một đoạn mã chính là một ảnh chụp gia tăng rời rạc, khởi tạo trước khi thực hiện đoạn mã và lưu trữ khi kết thúc đoạn mã đó. Các kết quả này được cập nhật (tiêm) vào không gian nhớ của tiến trình bằng các sử dụng hàm `inject` của checkpointer DICKPT. Chính điểm khác biệt này sẽ tạo nên ưu và nhược điểm của mỗi phương pháp. Khả năng tự động phân phối công việc, trích rút kết quả thực hiện và cập nhật kết quả vào trong không gian nhớ của các tiến trình tạo nên ưu điểm của CAPE ở tính trong suốt của quá trình trao đổi dữ liệu giữa các tiến trình đối với người lập trình và nhờ đó có thể tạo nên một cài đặt tương thích hoàn toàn với OpenMP. Tuy nhiên, việc sử dụng kỹ thuật đánh dấu tiến trình trong quá trình thực thi mã làm cho tốc độ thực hiện mã bị chậm lại và đây chính là nhược điểm của CAPE. Với MPI thì các đặc tính lại trái ngược với CAPE. Việc yêu cầu người sử dụng xác định tường minh các dữ liệu cần trao đổi, yêu cầu người lập trình phải viết các phần mã riêng biệt cho các tiến trình khác nhau cũng như yêu cầu người lập trình phải phân chia công việc cho các tiến trình làm cho MPI trở nên khó sử dụng, có khả năng gặp lỗi (do mã của người lập trình không đúng) trong khi chính điều này lại tạo nên hiệu suất hoạt động cao của kỹ thuật này.

4.2 Sự giống nhau

Ngoại trừ sự khác biệt nói trên, sự giống nhau của CAPE và MPI được thể hiện ở nhiều mặt bao gồm kiến trúc phần nền, mô hình tổ chức chương trình và mô hình thực hiện. Về kiến trúc phần nền, cả hai tuy vẫn có khả năng thực hiện trên các kiến trúc đa xử lý nhưng mục tiêu quan trọng hơn là nhắm tới các hệ thống sử dụng bộ nhớ phân tán chẳng hạn như cluster, lưới và đám mây. Về mô hình tổ chức chương trình, cả hai đều sử dụng kiểu đa tiến trình, trong đó có một tiến trình chính và một số tiến trình phụ, mỗi node của hệ thống có thể chạy một hoặc một số tiến trình. Về mô hình thực hiện, CAPE và MPI đều chứa những bước khá giống nhau. Để thực hiện một chương trình chứa các đoạn mã song song và tuần tự xen kẻ nhau, các bước tương ứng sẽ bao gồm phần khởi tạo chương trình, phần thực hiện các đoạn mã tuần tự và phần thực hiện mã song song trên một tập các tiến trình. Phần khởi tạo chương trình là giống nhau giữa CAPE và MPI, đều được thực hiện trên một tập các node, trong đó có một node chính và một node phụ. Đối với các đoạn mã tuần tự, vấn đề đặt ra là hoàn toàn giống nhau và có hai phương án: 1) thực hiện các đoạn mã này chỉ trên một tiến trình rồi cập nhật kết quả vào các tiến trình khác để đồng bộ không gian nhớ của chúng, và 2) thực hiện chúng một cách giống nhau trên tất cả các tiến trình và như vậy không cần bước đồng bộ hóa không gian nhớ của các tiến trình. Đối với phương án 1, vấn đề đặt ra tiếp theo là làm sao để trích rút và cập nhật kết quả thực hiện của một đoạn mã vào không gian nhớ của các tiến trình khác. Vấn đề này được xử lý một cách khác nhau trong MPI và CAPE, như được trình bày ở đoạn trên. Đối với các đoạn mã song song, cả MPI và CAPE đều bao gồm các bước con là phân chia công việc, thực thi phần tính toán song song ở các node tính toán và cập nhật kết quả về node chính. Tuy chứa các bước giống nhau như vậy, cách thức thực hiện các bước lại hoàn toàn khác nhau, một cách tự động trong CAPE và đòi hỏi người sử dụng viết mã tường minh để thực hiện, như đã trình bày ở đoạn trên.

Bảng 1 so sánh chi tiết các bước thực hiện một đoạn mã song song, dưới dạng chương trình MPI và CAPE. Như đã trình bày ở trên, các bước khởi tạo chương trình và thực hiện mã tuần tự là khá giống nhau giữa hai kỹ thuật nên không được đưa vào trong bảng này. Ở cột thứ nhất, ký hiệu t_i được ghi trong cặp ngoặc đơn là quy ước gọi thời gian thực hiện bước đó.

Bảng 1. Các bước thực hiện 1 đoạn mã song song của CAPE và MPI

Bước	**CAPE**		**MPI**	
	Cách thực hiện	**Ưu nhược điểm**	**Cách thực hiện**	**Ưu nhược điểm**
Phân chia công việc (t_b)	Tự động. Tiến trình chính tự động phân phối công việc đến các tiến trình phụ thông qua các ảnh chụp gia tăng rời rạc.	Trong suốt với người lập trình. Không tiêu tốn nhiều thời gian chạy chương trình.	Người lập trình viết mã để phân chia các tiến trình và phân chia công việc giữa các tiến trình.	Khó khăn cho người lập trình.
Thực thi phần mã tính toán (t_c)	Tiến trình phụ thực hiện mã của chương trình ứng dụng dưới sự theo dõi của một checkpointer DICKPT để trích rút kết quả ở bước sau.	Tiêu tốn thêm thời gian phụ do sự theo dõi của checkpointer trên tiến trình thực thi mã ứng dụng.	Tiến trình phụ thực hiện mã của chương trình ứng dụng một cách độc lập.	Hiệu năng cao.
Trích rút và gửi kết quả thực hiện về tiến trình chính	Tự động. Kết quả được trích rút dưới dạng ảnh chụp tiến trình gia tăng rời rạc.	Trong suốt với người lập trình. Trích rút chính xác kết quả thực hiện của một đoạn mã. Tiêu tốn thời gian phụ	Người lập trình viết mã để xác định dữ liệu cần gửi và nơi lưu trữ kết quả.	Hiệu năng cao nhưng có khả năng người lập trình xác định không chính

(t_u)		để kết xuất kết quả.		xác và bỏ sót kết quả.
Đồng bộ hóa không gian nhớ của các tiến trình (t_s)	Tự động. Không gian nhớ của tất cả các tiến trình được cập nhật bằng một ảnh chụp tiến trình gia tăng rời rạc chứa kết quả thực hiện phần mã song song trên tất cả các tiến trình.	Trong suốt với người lập trình. Việc đồng bộ được tiến hành một cách chính xác. Không tiêu tốn nhiều thời gian phụ so với việc cập nhật trực tiếp kiểu MPI.	Người lập trình viết mã để đồng bộ hóa.	Có khả năng bị bỏ sót và gặp lỗi do mã của người lập trình không đúng.

Trong một chương trình, lần lượt gọi t_{tt}, t_{ss} là thời gian thực hiện các đoạn mã tuần tự và song song của nó. Ta có thời gian thực hiện toàn bộ chương trình là:

$$t = \sum t_{tt_i} + \sum t_{ss_i} \qquad (4.2.1)$$

Với các quy ước gọi thời gian thực hiện các bước trong một đoạn mã song song như ở Bảng 1, ta có:

$$t = \sum t_{tt_i} + \sum (t_b + t_c + t_u + t_s)_i \qquad (4.2.2)$$

Thực tế thì việc song song hóa một phần mã nào đó chỉ cần thiết và có ý nghĩa thật sự khi thời gian thực hiện phần mã chính (tính toán) lớn hơn nhiều so với thời gian khởi tạo và trao đổi dữ liệu giữa các node cũng như thời gian đồng bộ hóa không gian nhớ của các tiến trình. Do đó thời gian thực hiện chương trình có thể được tính bằng công thức:

$$t = \sum t_{tt_i} + \sum (t_c + t_u)_i \qquad (4.2.3)$$

Với giả thiết các đoạn chương trình tuần tự được thực hiện trên tất cả các node với cả MPI và CAPE (do đó thời gian này là bằng nhau với cả hai kỹ thuật này), khi đó chênh lệch thời gian giữa CAPE và MPI sẽ là:

$$\Delta t = \sum \Delta (t_c + t_u)_i \qquad (4.2.4)$$

Về nguyên tắc, các đoạn mã tính toán trong phần mã song song là giống nhau giữa MPI và CAPE (giải thuật phân chia công việc là giống nhau), do đó sự khác biệt về thời gian trong t_c chính là do tác động của việc đánh dấu tiến trình kiểu gia tăng, trong đó tiến trình ứng dụng luôn bị giám sát bởi tiến trình checkpointer và do đó thời gian chạy mã ứng dụng bị tăng lên. Theo [15] thì số tăng này khoảng từ 2% đến 20%. Độ chênh lệch thời gian trong t_u gây ra do quá trình cập nhật qua các ảnh tiến trình là phức tạp hơn, tuy nhiên đối với những bài toán có t_c lớn thì t_u chiếm tỷ lệ khá bé trong tổng thời gian thực hiện chương trình và do đó độ chênh lệch thời gian ở bước này cũng chiếm tỷ lệ khá bé và có thể bỏ qua.

Từ phân tích ở trên, ta thấy sự chênh lệch về hiệu suất thực hiện chương trình giữa CAPE và MPI chủ yếu là do ảnh hưởng của quá trình đánh dấu tiến trình lên thời gian thực hiện mã ứng dụng ở các node tính toán. Để giảm sự chênh lệch này, có hai mặt cần chú ý. Một là cần cài đặt trình đánh dấu tiến trình (checkpointer) có hiệu năng cao, tức là ít tác động lên thời gian thực hiện mã ứng dụng của trình được chụp ảnh. Thứ hai là mã ứng dụng cần được phân chia sao cho giảm thiểu được kích thước ảnh chụp trên mỗi node, tức là giảm thiểu được vùng không gian nhớ được cập nhật trong tiến trình ứng dụng.

V. KẾT QUẢ THỰC NGHIỆM

Để kiểm chứng cho các nhận định trên, chúng tôi đã tiến hành thực nghiệm trên một bài toán nhân ma trận vuông, với kích thước thay đổi từ 3000x3000 đến 12000x12000. Các ma trận thuộc dạng bình thường với các phần tử mang giá trị khác 0. Chương trình được sử dụng gồm hai khối mã chính. Khối thứ nhất khởi tạo các ma trận và được thực hiện như nhau trên các node. Khối thứ hai gồm các vòng lặp `for` lồng nhau để thực hiện phép nhân ma trận. Đối với chương trình OpenMP và CAPE, vòng lặp `for` ngoài cùng được song song hóa bởi một `pragma parallel for` trong khi các bước lặp của nó được phân chia một cách tương ứng cho các node tính toán đối với chương trình MPI. Mã MPI được sử dụng ở đây đã được tối ưu hóa theo hướng giảm tối đa lượng dữ liệu trao đổi giữa các node. Hệ thống phần nền được sử dụng là một cluster các máy tính để bàn, mỗi máy trang bị CPU Intel®Core™2 Duo E8400 3GHz, RAM 2GB, chạy hệ điều hành Ubuntu 10.10, kết nối bằng mạng Ethernet tốc độ 100MB/s. Mỗi thử nghiệm được thực hiện ít nhất là 10 lần, với khoảng tin cậy tối thiểu là 90%. Các số liệu được đưa ra ở đây là trung bình cọng của 10 lần đo.

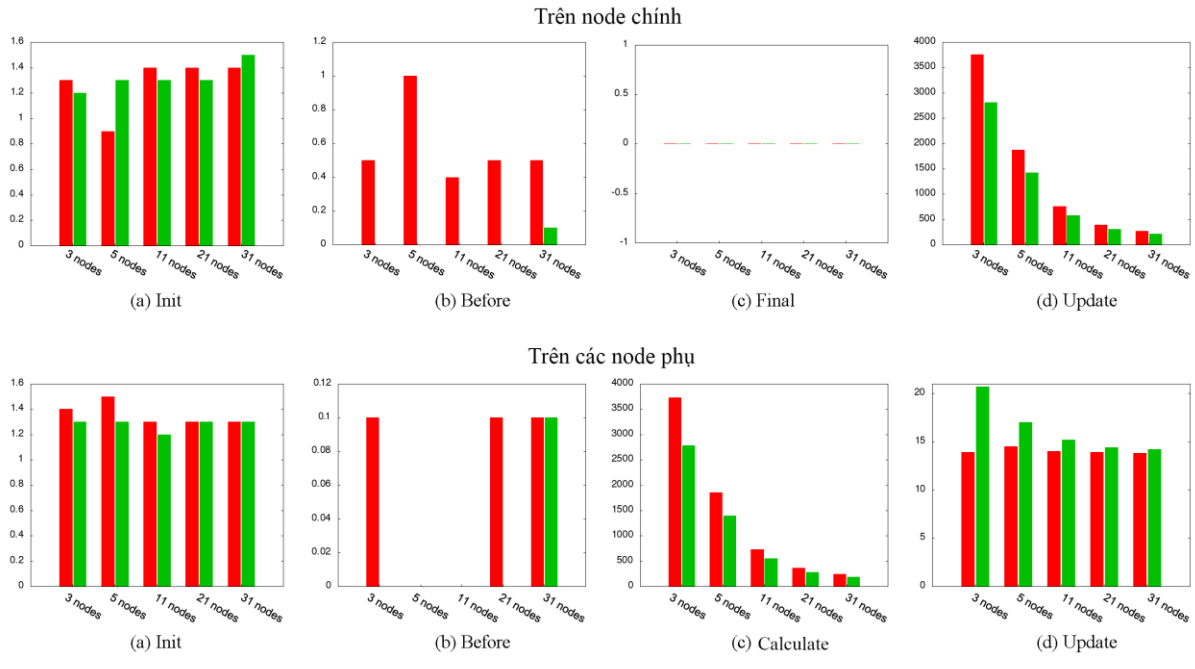
Bảng 2 trình bày thời gian thực hiện của chương trình dưới dạng tuần tự và dưới dạng OpenMP trên máy cục bộ với 2 tiến trình.

Bảng 2. Thời gian thực hiện trên một máy

Kích thước ma trận	Tuần tự (giây)	OpenMP (giây)
3000 x 3000	258,9	142,4
6000 x 6000	1852,7	1048,7
9000 x 9000	7314,5	3986,2
12000 x 12000	14990,5	8999,4

Ở Bảng 2, dễ dàng nhận thấy rằng thời gian thực thi của chương trình tuần tự và song song theo OpenMP tỷ lệ thuận với kích thước của ma trận. Dữ liệu cũng cho thấy, so với tốc độ thực thi của chương trình tuần tự, tốc độ thực thi của chương trình OpenMP nhanh gấp 1,8 lần đối với ba ma trận đầu tiên, và nhanh gấp 1,65 lần đối với ma trận thứ tư. Đây là các giá trị nằm trong dự kiến.

Phần tiếp theo của thử nghiệm là so sánh giữa CAPE và MPI.



Hình 4. Thời gian thực hiện theo số node (đơn vị tính: giây)

Hình 4 và Hình 5 thể hiện thời gian thực thi (đơn vị tính là giây) của việc nhân hai ma trận với số lượng các máy và kích thước ma trận khác nhau. Lưu ý rằng, mặt dù cấu hình máy tính được sử dụng để thực nghiệm là hai lõi (Dual Core), nhưng chỉ một lõi được sử dụng. Hai kỹ thuật được biểu diễn trong mỗi biểu đồ: cột bên trái biểu biểu diễn tốc độ thực hiện với CAPE, và bên phải biểu diễn tốc độ thực hiện với MPI. Tất cả các hình minh họa đều chứa hai nhóm biểu đồ. Nhóm phía trên liên quan đến các node chính, nhóm phía dưới liên quan đến node phụ. Các chỉ số về dòng mã được tham chiếu từ Hình 3. Mỗi nhóm biểu đồ lại chứa 4 biểu đồ thành phần:

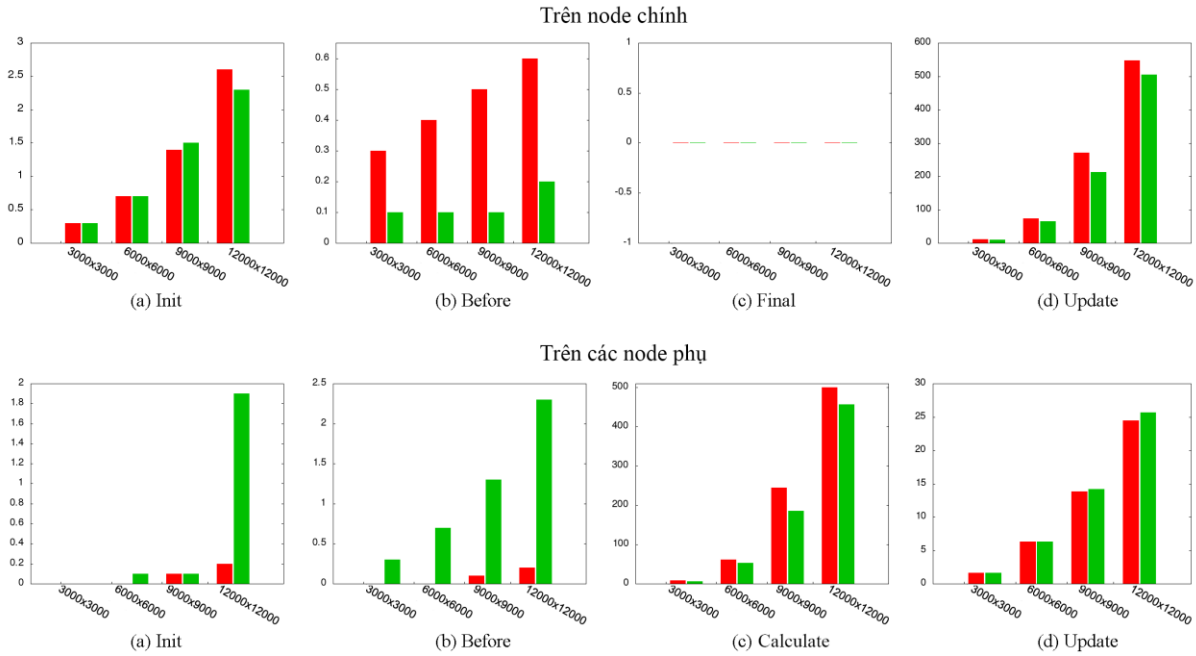
Init: là khoảng thời gian từ khi bắt đầu chạy chương trình đến khi bắt đầu thực hiện vòng lặp `for` song song trong bài toán nhân 2 ma trận. Theo quy ước ở công thức 4.2.1, t_{tt} là thời gian thực hiện của bước này.

Before: với CAPE, trên node chính, là khoảng thời gian để tạo ra và truyền các ảnh chụp tiến trình đến các node phụ (dòng 2-5). Đối với các node phụ, thời gian này bao gồm việc nhận và cập nhật ảnh chụp tiến trình vào không gian nhớ của tiến trình hiện tại (dòng 14-15). Đối với MPI, đây là thời gian truyền dữ liệu khởi tạo từ node chính đến các node con. Theo quy ước ở Bảng 1, t_b là thời gian thực hiện của bước này.

Final/Calculate: là thời gian tạo ra ảnh chụp tiến trình cuối cùng trên node chính (dòng 6-7) và thời gian để thực hiện việc tính toán trên các node phụ (dòng 16-17). Theo quy ước ở Bảng 1, t_c là thời gian thực hiện của bước này.

Update: là thời gian để node chính nhận tất cả các kết quả tính toán từ các node phụ và cập nhật vào không gian nhớ của node chính (dòng 8-9). Trên các node phụ, đây là khoảng thời gian để tạo các ảnh chụp tiến trình và gửi chúng đến node chính (dòng 18-20). Đối với MPI, đây là khoảng thời gian gửi kết quả tính toán được từ các node phụ đến node chính. Theo quy ước ở Bảng 1, t_u là thời gian thực hiện của bước này.

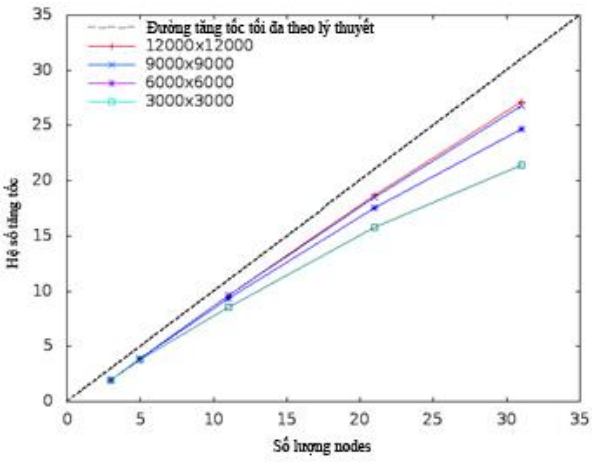
Hình 4 biểu diễn thời gian thực hiện với số lượng số node khác nhau trên bài toán nhân 2 ma trận có kích thước 9000x9000. Tuy vậy, đối với các ma trận có kích thước khác, các kết quả có xu hướng tương tự. Các biểuđồ cho thấy trên node chính, các pha *Init, Before* và *Final* có thời gian rất nhỏ so với bước *Update* (t_{tt}, t_b và t_c rất nhỏ so với t_u). Điều này là do bước *Update* bao gồm trong đó việc chờ và nhận các kết quả tính toán từ các node phụ, tức là chứa cả quảng thời gian các mà các đoạn mã tính toán việc nhân ma trận được thực hiện. Từ nhận xét này, ta có thể xem thời gian thực hiện chương trình là bằng t_u trên node chính. Tương tự như vậy đối với các node phụ, thời gian của các bước *Init* và *Before* có thể được bỏ qua và có thể xem thời gian thực hiện chương trình là tổng của thời gian thực hiện việc phần mã tính toán và cập nhật kết quả thực hiện. Các nhận xét này phù hợp với các công thức 4.2.3 và 4.2.4.



Hình 5. Thời gian thực hiện theo kích thước ma trận (đơn vị tính: giây)

Dữ liệu trên Hình 4 cũng cho thấy, với thử nghiệm trên 3 node, thời gian thực hiện CAPE luôn luôn lớn hơn rất nhiều thời gian thực hiện của MPI. Kết quả này là do lúc này mỗi node tính toán thực hiện một nửa số tính toán của phép nhân ma trận và cũng cập nhật một nửa kích thước ma trận kết quả, do đó kích thước của ảnh chụp tiến trình chiếm tỷ lệ khá lớn trong tổng không gian nhớ của tiến trình, dẫn đến tác động của quá trình đánh dấu tiến trình lên thời gian thực thi mã ứng dụng là lớn. Tuy nhiên khi tăng dần số lượng máy, tương ứng với nó là các ảnh chụp tiến trình có kích thước càng nhỏ thì khoảng cách chênh lệch giữa CAPE và MPI càng lúc càng thu hẹp. Khi tăng đến 31 máy, thời gian này là xấp xỉ như nhau. Kết quả này cho thấy hiệu suất của CAPE rất kém với số node ít và chỉ nên sử dụng nó với số node khá lớn. Tuy nhiên, điều này là không nghiêm trọng vì hiện tại các máy tính với bộ xử lý 4 lõi đã trở nên phổ biến, và số lõi sẽ ngày càng tăng. Do đó, với yêu cầu số tiến trình thấp, thay vì sử dụng CAPE, ta có thể sử dụng OpenMP kiểu đa thread trên các máy tính đa lõi này. Mặt khác, nó cũng chỉ ra rằng, CAPE có khả năng đạt hiệu suất cao khi kích thước các ảnh chụp tiến trình là bé, tức là các đoạn mã ứng dụng tại các node phụ chỉ cập nhật một dung lượng nhỏ không gian nhớ của tiến trình.

Hình 5 biểu diễn thời gian thực thi đối với các kích thước ma trận khác nhau. Số lượng máy để thực hiện tính toán song song là 31 máy. Tuy vậy, các kết quả có xu hướng tương tự khi thực nghiệm trên hệ thống có số lượng máy tính khác nhau (trừ trường hợp 3 node). Các nhận xét cho biểu đồ ở Hình 4 cũng đúng cho biểu đồ này. Số liệu trên biểu đồ cho thấy thời gian thực thi của CAPE và MPI tỷ lệ thuận với kích thước của ma trận. Do cả MPI và CAPE đều sử dụng cơ chế chỉ truyền dữ liệu để cập nhật kết quả tính toán, nên tốc độc thực thi cũng khá tương tự nhau. Phân tích chi tiết hơn, ta thấy rõ tổng thời gian thực thi của CAPE sử dụng kỹ thuật đánh dấu tiến trình gia tăng rời rạc chỉ cao hơn khoảng 10% so với của MPI, trừ ma trận có kích thước 3000x3000 có tỷ lệ là 1,3.



Hình 6. Hệ số tăng tốc của CAPE theo số node

Hình 6 biểu diễn hệ số tăng tốc (speedup) của CAPE trên số lượng các máy và các kích thước ma trận khác nhau. Đường đứt nét biểu diễn sự gia tăng tối đa lý thuyết. Biểu đồ này cho thấy một cách rõ ràng rằng giải pháp CAPE đã đạt được kết quả rất tốt, với hệ số tăng tốc đo được nằm trong khoảng 75% đến 90%.

VI. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

CAPE, với khả năng tự động phân chia công việc và xác lập các dữ liệu trao đổi giữa các tiến trình, đã chứng tỏ được khả năng của nó để cài đặt OpenMP một cách tương thích hoàn toàn trên hệ thống sử dụng bộ nhớ phân tán. Mặt khác, các kết quả so sánh hiệu suất của CAPE với MPI thông qua phân tích lý thuyết cũng như thực nghiệm ở các chương trên đã chứng tỏ được về mặt hiệu suất thì CAPE hoàn toàn có khả năng so sánh được với giải pháp có hiệu năng cao nhất là MPI, nhất là với hệ thống có số node lớn và các chương trình có mã sao cho sau khi phân chia thì ở mỗi node, dung lượng không gian nhớ được cập nhật là bé (so với toàn bộ không gian nhớ của tiến trình). Các kết quả ở trên cũng cho thấy nếu yêu cầu chạy OpenMP với số thread bé thì nên sử dụng OpenMP kiểu đa thread (thông qua một trình biên dịch C/C++ hỗ trợ OpenMP, chẳng hạn *gcc* trên Linux) trên một máy tính có bộ xử lý đa lõi. Trường hợp cần song song hóa với số lượng lớn các thread nhưng các mã thành phần cập nhật một phần lớn không gian nhớ của tiến trình thì nên sử dụng MPI, mặc dù chương trình ở dạng này khó viết hơn.

Kế hoạch ngắn hạn tiếp theo của chúng tôi là thử nghiệm CAPE với tập các bài toán phong phú hơn. Tiếp theo chúng tôi sẽ tiếp tục phát triển CAPE để nó vượt qua các ràng buộc của điều kiện Bernstein đối với các bài toán ứng dụng.

Với kế hoạch dài hạn hơn, chúng tôi sẽ nghiên cứu các mô hình hoạt động khác của CAPE để nó có thể được sử dụng một cách linh hoạt hơn, chẳng hạn như cho phép các đoạn mã tuần tự của chương trình ứng dụng chỉ cần được thực hiện trên một tiến trình, hoặc việc cập nhật không gian nhớ của các tiến trình phụ ở cuối mỗi đoạn mã song song có thể được làm trễ để có thể kết hợp với phần cập nhật kết quả của đoạn mã tuần tự, điều này nhằm tăng hiệu suất hoạt động tổng thể của chương trình.

TÀI LIỆU THAM KHẢO

[1] Message Passing Interface Forum, http://www.mpi-forum.org/

[2] The OpenMP® API specification for parallel programming, http://openmp.org/wp/

[3] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, R. Badrinath and Louis Rilling.,"Kerrighed: A Single System Image Cluster Op-erating System for High Performance Computing", Euro-Par 2003 Parallel Pro-cessing, Klagenfurt, Austria, LNCS 2790, pp. 1291–1294(2003).

[4] Mitsuhisa Sato, Hiroshi Harada, Atsushi Hasegawa and Yutaka Ishikaw,"Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software dis-tributed shared memory system", Journal Scientific Programming, Volume 9 Issue 2,3 (2001).

[5] Sven Karlsson, Sung-Woo Lee, Mats Brorsson, Sahni Sartaj, Viktor K. Prasanna and Shukla Uday, "A fully compliant OpenMP implementation on software dis-tributed shared memory". Proceedings of the International Conference on High Performance Computing, Bangalore, India, LNCS 2552, pp. 195–206 (2002).

[6] Ayon Basumallik and Rudolf Eigenmann, "Towards automatic translation of OpenMP to MPI", Proceedings of the 19th annual international conference on Supercomputing, Cambridge, MA, pp. 189–198 (2005).

[7] Beniamino Di Martino, Dieter Kranzlmüller and Jack Dongarra, "Implementing OpenMP for clusters on top of MPI", Proceedings of 12th European PVM/MPI Users' Group Meeting Sorrento, LNCS, Volume 3666, pp 148–155 (2005).

[8] Lei Huang and Barbara Chapman and Zhenying Liu, "Towards a more efficient implementation of OpenMP for clusters via translation to global arrays", Journal of Parallel Computing 31, pp 1114–1139(2005).

[9] Jay P. Hoeflinger, "Extending OpenMP* to Clusters", White paper, Available at: http://140.110.240.196/grid/raw-attachment/wiki/Osaka/ In-tel_Extend_OpenMP_Cluster.pd

[10] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, PascalGallard, Gaël Utard, R. Badrinath and Louis Rilling, "Kerrighed: A Single System Image Cluster Operating System for High Performance Computing", Euro-Par 2003 Parallel Processing, Klagenfurt, Austria, LNCS 2790, pp. 1291–1294 (2003).

[11] Eric Renault, "Distributed Implementation of OpenMP Based on Checkpointing Aided Parallel Execution", International Workshop on OpenMP (IWOMP), Beijing, China, LNCS 4935, pp.183-193(2007).

[12] Viet Hai Ha and Éric Renault, "Improving Performance of CAPE using Discontinuous Incremental Checkpointing", Proceedings of the International Conference on High Performance and Communications 2011 (HPCC-2011), Banff, Canada (2011).

[13] Jame S. Plank, "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance", Technical Report UTCS-97-372, Department of Computer Science, University of Tennessee, Juilly (1997).

[14] Viet Hai Ha and Éric Renault, "Discontinous Incremental: A New Approach Towards Extremely Lighweight Checkpoints", Proceedings of the IEEE Incremental Symposium on Computer Networks and Distributed System 2011 (CNDS 2011), Tehran, Iran (2011).

[15] Gioiosa R., Sancho J.C., Jiang S. and Petrini F, "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers", Proceedings of the ACM/IEEE SC 2005 Conference (2005).

ANALYSE AND EVALUATE THE PERFORMANCE OF CAPE

Viet Hai Ha[1], Van Long Tran[2]

[1] Hue University's College of Education, Vietnam
[2] Hue Industrial College, Vietnam
haviethai@gmail.com, tvlong@hueic.edu.vn

ABSTRACT— *MPI (Message Passing Interface) and OpenMP are two tools broadly used to develop parallel programs. MPI has advantage of high performance but is difficult to be used while OpenMP is very easy to be applied. The most important disadvantage of OpenMP is its restriction on shared memory architectures. CAPE (Checkpointing Aided Parallel Execution) is an approach using checkpointing to implement OpenMP on distributed memory architectures. This paper aims at deeply analyzing and evaluating the performance of CAPE by comparing its execution model and the one of MPI, then providing the suggestions to use CAPE effectively.*