ĐẠI HỌC HUẾ

TRƯỜNG ĐẠI HỌC KHOA HỌC

ĐỖ XUÂN HUYỀN

CẢI TIẾN MÔ HÌNH CAPE CHO HỆ THỐNG TÍNH TOÁN ĐA LÕI

LUẬN ÁN TIẾN SĨ KHOA HỌC MÁY TÍNH

HUẾ - NĂM 2023

ĐẠI HỌC HUẾ

TRƯỜNG ĐẠI HỌC KHOA HỌC

ĐỖ XUÂN HUYỀN

CẢI TIẾN MÔ HÌNH CAPE CHO HỆ THỐNG TÍNH TOÁN ĐA LÕI

NGÀNH: KHOA HỌC MÁY TÍNH

MÃ SỐ: 9480101

LUẬN ÁN TIẾN SĨ KHOA HỌC MÁY TÍNH

Người hướng dẫn khoa học

1. TS. HÀ VIẾT HẢI

2. GS. ÉRIC RENAULT

HUẾ - NĂM 2023

LỜI CAM ĐOAN

Tôi xin cam đoan đây là công trình nghiên cứu do tôi thực hiện dưới sự hướng dẫn của TS. Hà Viết Hải và GS. Éric Renault. Những nội dung trong các công trình đã được công bố chung với các tác giả khác đã được sự chấp thuận của đồng tác giả khi đưa vào luận án. Các số liệu và kết quả nghiên cứu được trình bày trong luận án là trung thực, khách quan và chưa được công bố bởi tác giả nào trong bất kỳ công trình nào khác.

Nghiên cứu sinh

Đỗ Xuân Huyền

LỜI CẢM ƠN

Trước hết tôi xin bày tỏ lòng biết ơn chân thành và sâu sắc đến TS. Hà Viết Hải và GS. Éric Renault là những người Thầy đã tận tình hướng dẫn, chỉ bảo, động viên và giúp đỡ để tôi có thể hoàn thành được luận án này.

Tôi xin trân trọng cảm ơn sự giúp đỡ của Quý Thầy Cô trong Khoa Công nghệ Thông tin, Trường Đại học Khoa học Huế đã quan tâm, giúp đỡ, hướng dẫn trong suốt quá trình học tập.

Tôi xin trân trọng cảm ơn Các đồng nghiệp, Ban Giám đốc, Trung tâm Công nghệ Thông tin Thừa Thiên Huế đã tạo điều kiện thuận lợi trong công tác để tôi có đủ thời gian hoàn thành luận án này.

Tôi xin trân trọng cảm ơn Ban Giám hiệu, Quý Thầy Cô trong Khoa Tin học, Trường Đại học Sư phạm, Đại học Huế; Quý Thầy Cô, cán bộ quản lý Phòng Đào tạo Sau đại học, Trường Đại học Khoa học, Đại học Huế đã giúp đỡ tôi hoàn thành kế hoạch học tập.

Cuối cùng tôi xin chân thành cảm ơn các bạn bè đồng nghiệp, người thân trong gia đình luôn động viên, giúp đỡ tôi về mọi mặt trong suốt quá trình nghiên cứu, học tập của mình.

Nghiên cứu sinh

Đỗ Xuân Huyền

MỤC LỤC

DANH MỤC CÁC TỪ VIẾT TẮT

Từ viết tắt	Thuật ngữ tiếng Anh	Thuật ngữ/dịch tiếng Việt
API	Application Programming Interface	Giao diện lập trình ứng dụng
CAPE	Checkpointing Aided Parallel Execution	Chạy chương trình song song dựa trên kỹ thuật chụp ảnh tiến trình
CPU	Central Processing Unit	Bộ xử lý trung tâm
DICKPT	Discontinuos Incremental Checkpointing	Chụp ảnh tiến trình gia tăng rời rạc
DSM	Distributed Shared Memory	Bộ nhớ chia sẻ phân tán
EP	Entire page	Cấu trúc toàn trang nhớ
File	File	Tập tin / Tệp tin
GPU	Graphics Processing Unit	Bộ xử lý các tác vụ có liên quan tới đồ họa
HLRC	Home-based Lazy Release Consistency	Mô hình đồng bộ trễ đồng nhất
HPC	High performance computing	Tính toán hiệu năng cao
ICKPT	Incremetal Checkpointing	Kỹ thuật chụp ảnh tiến trình gia tăng
KVM	Kernel-based Virtual Machine	Máy ảo dựa trên mức nhân (kernel)
LAN	Local Area Network	Mạng cục bộ
MMU	Memory Management Unit	Bộ quản lý bộ nhớ
MD	Many data	Cấu trúc sơ đồ ma trận
MPI	Message Passing Interface	API lập trình song song truyền thông điệp
Multi-processing	Multi-processing	Đa tiến trình, đa chương trình
Multi-threading	Multi-threading	Đa luồng, đa tiểu tiến trình
OpenMP	Open Multi-Processing	API lập trình song song trên hệ thống SM
OS	Operating System	Hệ điều hành

Page table	Page table	Bảng danh mục trang
RC	Relaxed-Consistency	Chia sẻ dữ liệu đồng bộ trễ
SD	Single data	Cấu trúc đơn từ
SIGSEGV	Signal segmentation violation	Tín hiệu dạng ngắt của OS.
SM	Shared Memory	Bộ nhớ chia sẻ
SSD	Several successive data	Cấu trúc dữ liệu liên tiếp
SSI	Single System Image	Một hệ thống các máy tính được liên kết với nhau (ảo hóa) tạo thành một máy thống nhất với bộ nhớ chung
TLB	Translation Lookaside Buffer	Bộ nhớ cache cho Page Tables

.

DANH MỤC CÁC HÌNH VẼ

DANH MỤC CÁC BẢNG

MỞ ĐẦU

Tính cấp thiết của đề tài

Đồng hành với việc tăng tốc độ xử lý của các bộ vi xử lý, yêu cầu về tốc độ tính toán của các bài toán lớn cũng không ngừng tăng theo. Do vậy, luôn có những bài toán cần tốc độ xử lý vượt quá tốc độ xử lý của một chương trình tuần tự. Điều này càng đúng trong thời điểm hiện tại, khi các kiến trúc vi xử lý đã gần đạt đến giới hạn cuối cùng về kích thước của đơn vị cơ sở cũng như xung nhịp tối đa, cũng đồng nghĩa với việc tăng tốc độ của các một bộ xử lý đơn lõi lên gấp đôi sau 24 tháng– theo định luật Moore[1] – sau gần 50 năm nghiệm đúng đã đến tiến gần thời điểm kết thúc. Do đó, giải pháp chính hiện tại và trong tương lai gần về mặt phần cứng cho việc tăng tốc độ tính toán là tăng số lõi của bộ xử lý. Điều này càng làm tăng thêm tầm quan trọng của xử lý song song (parellel processing) để một chương trình có khả năng khai thác tối đa khả năng của các lõi trên bộ vi xử lý đa lõi, như là một hướng giải quyết khả thi hiệu quả và ít tốn kém để đáp ứng cho những nhu cầu tính toán tốc độ cao.

Nguyên lý chung của xử lý song song là chia bài toán lớn (theo số lần thực hiện các câu lệnh hoặc theo kích thước dữ liệu hoặc đồng thời theo cả hai) thành nhiều phần con và cho thực hiện một cách song song các phần này trên các đơn vị xử lý khác nhau. Nguyên lý này khác biệt căn bản với nguyên lý hoạt động của mô hình xử lý tuần tự (sequential processing), thực hiện lần lượt các xử lý, xử lý sau chỉ tiến hành khi đã xong xử lý trước.

Tuy có ưu điểm về mặt tốc độ tính toán, mô hình xử lý song song cũng có những nhược điểm nhất định. Thứ nhất, xử lý song song đòi hỏi phải có những kiến trúc phần nền (cả phần cứng và phần mềm) hỗ trợ. Thứ hai, có công cụ lập trình khai thác khả năng của phần nền này để tạo giao diện cho người lập trình có thể tạo các chương trình xử lý song song. Yêu cầu thứ nhất được đáp ứng qua các mô hình đa bộ xử lý và/hoặc bộ xử lý đa lõi (trên CPU và gần đây là GPU) cùng với các kiến trúc mạng. Yêu cầu thứ hai cũng đã liên tục phát triển tương ứng với các phát triển của phần nền mà điển hình như các mô hình đa tiến trình (multi-processing), đa luồng (đa tiểu tiến trình, multi-

[1] Định luật Moore lần đầu tiên được công bố rộng rãi trên tạp chí Electronics Magazine số ra ngày 19 tháng 4 năm 1965. Định luật Moore (tiếng Anh: Moore's Law) đề cập đến dự đoán của Gordon E. Moore, nhà đồng sáng lập hãng Intel, rằng số lượng bóng bán dẫn trên một vi mạch sẽ tăng gấp đôi sau mỗi 24 tháng".

threading), truyền thông điệp (message passing) và mới đây là các công cụ ở mức trừu tượng cao hơn như OpenMP, MPI, Map Reduce, Cuda và OpenCL.

Đối với cách sử dụng đa tiến trình mỗi tiến trình hoạt động tương độc lập với các tiến trình khác và có không gian bộ nhớ riêng. Mô hình này có ưu điểm, nếu tiến trình bị sự cố sẽ không làm ảnh hưởng đến các tiến trình khác. Nhược điểm của mô hình đa tiến trình thể hiện ở sự phức tạp và tốc độ thấp của việc chia sẻ và đồng bộ hóa dữ liệu. Ngược lại, nếu thực hiện chương trình song song theo mô hình đa luồng, trong đó các luồng sử dụng bộ nhớ chia sẻ chung, việc chia sẻ dữ liệu và đồng bộ dữ liệu trở nên rất đơn giản, với tốc độ cao, nhưng khi một luồng gặp sự cố giữa chừng sẽ kéo theo toàn bộ chương trình đổ vỡ.

Tuy việc lập trình song song với các mô hình đa tiến trình, đa luồng đã được các ngôn ngữ lập trình phổ dụng như C/C++, C#, Java, Fortran,… đáp ứng nhưng những ngôn ngữ này đòi hỏi nhiều công sức cũng như kiến thức, kinh nghiệm của lập trình viên. Do lập trình viên phải tự mình viết mã cho những công việc trung gian để tạo ra chương trình song song như như việc sinh ra các tiến trình, luồng, phân công công việc cho mỗi đơn vị thực hiện đó, đảm bảo đồng bộ hóa dữ liệu, thu thập dữ liệu từ các đơn vị thực hiện song song... Do đó, mô hình lập trình song song cần có một công cụ lập trình ở mức trừu tượng cao hơn, giúp tự động hóa xử lý các công việc trung gian để giúp các lập trình dễ học, dễ lập trình và có thể dễ dàng mang lại hiệu năng cao. Hiện tại, hai công cụ phổ dụng được sử dụng cho hai mô hình đa luồng và đa tiến trình là OpenMP và MPI [65].

OpenMP [1] là một API cung cấp một mức trừu tượng hóa cao để viết các chương trình song song cho các hệ thống sử dụng bộ nhớ chia sẻ (máy tính sử dụng CPU multi-core và/hoặc multi-CPU). 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 C/C++ hoặc Fortran thành một chương trình song song. OpenMP sử dụng mô hình thực hiện *fork-join* với cấu trúc song song cơ sở là luồng. Do sử dụng cấu trúc cơ sở là luồng, mặc nhiên mô hình bộ nhớ của OpenMP là bộ nhớ chia sẻ, trong đó không gian nhớ được sử dụng chung giữa các luồng. Để viết chương trình song song với OpenMP, lập trình viên có thể bắt đầu bằng cách viết một chương trình tuần tự trên ngôn ngữ gốc (C/C++ hoặc Fortran), sau đó thêm dần vào các chỉ thị của OpenMP để chỉ định những

phần việc nào cần được thực hiện song song. Việc chia sẻ dữ liệu cũng như đồng bộ dữ liệu được tiến hành một cách ngầm định hoặc tường minh qua các chỉ thị đơn giản. Nhờ đặc điểm này mà OpenMP trở nên dễ học, dễ sử dụng và ít tốn công sức lập trình nhưng lại có thể cung cấp hiệu năng cao trên các kiến trúc sử dụng bộ nhớ chia sẻ. Tuy nhiên, OpenMP vẫn chỉ mới được cài đặt một cách hoàn chỉnh cho các kiến trúc sử dụng bộ nhớ chia sẻ do sự phức tạp của việc cài đặt tất cả các yêu cầu của OpenMP trên các kiến trúc sử dụng bộ nhớ khác.

MPI (Message Passing Interface) [4] là chuẩn giao tiếp trong các ứng dụng tính toán hiệu năng cao (High performance computing - HPC) chủ yếu nhắm đến việc lập trình song song trên các máy tính phân tán, sử dụng đa tiến trình với mô hình bộ nhớ phân tán. Một trong những đặc điểm chính của các chương trình MPI là lập trình viên phải tự mình tổ chức chương trình song song theo kiến trúc của MPI cũng như viết các câu lệnh tường minh để trao đổi và đồng bộ dữ liệu giữa các tiến trình trên các nút (node/máy tính) của hệ thống. Điều này giúp giảm dung lượng dữ liệu giao dịch trên mạng và làm cho MPI trở thành môi trường lập trình song song có hiệu năng cao nhất trên kiến trúc bộ nhớ phân tán. Tuy nhiên, MPI lại đòi hỏi lập trình viên phải tốn nhiều công sức lập trình hơn vì phải viết các câu lệnh tường minh để tổ chức chương trình, trao đổi dữ liệu giữa các tiến trình. Lập trình viên cũng phải thiết kế một cách tường minh cấu trúc của chương trình song song, một điều khó hơn so với thiết kế một chương trình tuần tự.

Cả OpenMP và MPI đều có ưu và nhược điểm riêng. Ưu điểm lớn nhất của OpenMP là tính dễ học, dễ sử dụng đối với lập trình viên và có hiệu năng cao trên các hệ thống sử dụng bộ nhớ chia sẻ còn MPI là khả năng cung cấp hiệu năng cao trên hệ thống bộ nhớ phân tán. Nhược điểm của OpenMP là chưa được hỗ trợ hoàn toàn trên hệ thống bộ nhớ phân tán, trong khi của MPI là tính khó học, khó và tốn công sức khi lập trình. Điều này dẫn đến một ý tưởng và động lực nghiên cứu để chuyển đổi OpenMP lên các kiến trúc sử dụng bộ nhớ phân tán nhằm phát huy ưu điểm và khắc phục nhược điểm của chúng.

Thực hiện ý tưởng trên, trong nhiều năm qua, đã có nhiều nhóm nghiên cứu cố gắng thực hiện việc đưa OpenMP trên hệ thống máy tính sử dụng bộ nhớ phân tán bằng cách xây dựng trình biên dịch để dịch tự động chương trình OpenMP thành chương trình

có khả năng chạy trên các hệ thống này. Đồng thời với việc xây dựng các chương trình biên dịch, một số tiếp cận còn đòi hỏi phải xây dựng một nền tảng (platform) bổ sung cho hệ thống để có thể chạy được các chương trình đã được biên dịch. Tuy nhiên, ngoại trừ CAPE [12]-[19], chưa có công trình nào thành công trên cả hai mặt, tương thích hoàn toàn với OpenMP và có hiệu năng cao. Một số công trình nghiên cứu nổi bật có thể nhắc đến như SSI [6]; Cluster OpenMP [11]; SCASH [7]; sử dụng mô hình HLRC [24]; biên dịch thành MPI [8][9]; sử dụng Global Array [10]; libMPNode [30] cải tiến SSI; OMPC [34] sử dụng trình biên dịch riêng. Hiện tại, OpenMP nói chung và phát triển OpenMP trên các hệ thống sử dụng bộ nhớ phân tán nói riêng là chủ đề chính của chuỗi hội thảo IWOMP, với lần thứ 19 sẽ được tổ chức tại Đại học Bristol, Anh vào tháng 9 năm 2023.

CAPE (Checkpointing Aided Parallel Execution) là một tiếp cận dựa trên kỹ thuật chụp ảnh tiến trình để cài đặt API OpenMP trên các hệ thống máy tính sử dụng bộ nhớ phân tán. Chụp ảnh tiến trình (checkpointing) là một kỹ thuật cơ bản được sử dụng chủ yếu trong việc phục hồi trạng thái chương trình ở các thời điểm chạy trước đó. Kỹ thuật này được thực hiện việc này bằng cách lưu trạng thái chương trình đang chạy ở nhiều thời điểm khác nhau vào các file ảnh tiến trình và sử dụng các file ảnh tiến trình này để phục hồi lại trạng thái chương trình đã chạy trước đó khi cần mà không cần phải tốn thời gian chạy lại chương trình từ đầu.

Nguyên lý hoạt động của hiện tại CAPE sử dụng đơn vị song song cơ sở là tiến trình (process), thay cho luồng (thread) như trong OpenMP nguyên bản để có thể vận hành trên các hệ thống nhớ phân tán. Có thể hiểu khái quát nguyên lý hoạt động của CAPE như sau: 1) Trạng thái và kết quả của chương trình chạy trên hệ điều hành được thể hiện thông qua trạng thái bộ nhớ và trạng thái thanh ghi của chương trình và trạng thái này có thể theo dõi, lưu giữ được nhờ kỹ thuật chụp ảnh tiến trình; 2) CAPE sử dụng mô hình phân bố theo kiến trúc chính-phụ (master-slave) với quy trình căn bản như sau: đối với mỗi phần chương trình song song OpenMP, nút chính phân chia công việc thành phần cho các tiến trình ở trên các nút phụ trong hệ thống; các nút phụ nhận nhiệm vụ từ nút chính và thực hiện các công việc tính toán, sau đó gửi kết quả ở dạng ảnh chụp tiến trình về cho nút chính; nút chính nhận kết quả và tích hợp vào không gian nhớ của nút chính và xem như đã ở trạng thái hoàn thành đoạn chương trình tính toán song song.

Động lực nghiên cứu

CAPE đã được chứng minh qua các kết quả nghiên cứu lý thuyết cũng như số liệu thực nghiệm, trong đó chứng tỏ hệ số tăng tốc cũng như khả năng mở rộng (scale up) đối với các chương trình song song của CAPE rất tốt [13][17], có thể so sánh với MPI – kỹ thuật cung cấp hiệu năng cao nhất trên hệ thống bộ nhớ phân tán. Tuy nhiên, trong mô hình thực hiện của CAPE hiện tại, mỗi nút phụ thực hiện các công việc tính toán chỉ chạy một tiến trình thực hiện một phần nhiệm vụ song song do nút chính phân phối. Điều này trở nên lãng phí tài nguyên khi sử dụng CAPE trên hệ thống máy tính có CPU đa lõi rất phổ biến hiện nay. Để khắc phục được hạn chế này, mô hình thực hiện của CAPE cần phải được tổ chức theo hướng cho phép chương trình chạy song song trên từng nút phụ (song song mức thứ 2) để khai thác tốt hơn tài nguyên hệ thống và từ đó tăng tốc độ tính toán. Đây chính là động lực để luận án đề xuất mô hình CAPE mới khắc phục những hạn chế chưa khai thác tài nguyên hệ thống máy tính sử dụng CPU đa lõi đã trở nên phổ dụng hiện tại. Để thực hiện mục tiêu này những vấn đề sau cần được tiếp tục phát triển:

• Phát triển một mô hình CAPE mới để song song hóa một cách hiệu quả các đoạn mã tính toán trên từng nút tính toán (nút phụ);

• Phát triển kỹ thuật chụp ảnh tiến trình cũng như giải quyết vấn đề chia sẻ đồng bộ dữ liệu của CAPE cho phù hợp với các mô hình thực hiện được song song hóa 2 mức ở mục trên.

Nếu các yêu cầu trên được thực hiện tốt, dự đoán hệ số tăng tốc của CAPE có thể tăng với một hệ số tuyến tính với số lõi của các bộ xử lý trên các máy tính toán trong hệ thống. Điều này có nghĩa có thể tăng khá nhiều đối với các bộ xử lý với 4 đến 8 lõi hiện nay và nhiều hơn cho xu thế số lõi của các bộ xử lý được tăng đều trong tương lai gần.

Từ những trình bày trên, đề tài "Cải tiến mô hình CAPE cho hệ thống tính toán đa lõi" trở nên có tính thời sự và cấp thiết để đáp ứng nhu cầu cung cấp một giải pháp cài đặt tương thích hoàn toàn OpenMP và có hiệu năng cao trên các hệ thống bộ nhớ phân tán.

Mục tiêu nghiên cứu

Phát triển mô hình CAPE trên hệ thống tính toán đa lõi để nâng cao hiệu năng hoạt động của hệ thống. Cụ thể:

- Nghiên cứu và đề xuất mô hình hoạt động mới của CAPE trên hệ thống tính toán đa lõi;

- Nghiên cứu và xây dựng phiên bản kỹ thuật chụp ảnh đa tiến trình phù hợp với mô hình hoạt động mới được đề xuất trên hệ thống tính toán đa lõi;

- Nghiên cứu và đề xuất giải pháp xử lý các vấn đề chia sẻ dữ liệu của CAPE trên hệ thống tính toán đa lõi;

- Phát triển hệ thống phần mềm tương ứng với mô hình hoạt động của CAPE mới được đề xuất và đánh giá hiệu năng của CAPE mới so với mô hình CAPE trước đó và với MPI (kỹ thuật cung cấp hiệu năng tốt nhất hiện nay trên các hệ thống bộ nhớ phân tán).

Đối tượng và phạm vi nghiên cứu

- *Đối tượng nghiên cứu:* CAPE, OpenMP, phương thức hoạt động của bộ nhớ ảo của hệ điều hành, CPU đa lõi.

- *Phạm vi nghiên cứu:* CAPE áp dụng trên các cluster với các máy tính cá nhân sử dụng bộ vi xử lý đa lõi.

Phương pháp nghiên cứu

Luận án sử dụng hai phương pháp nghiên cứu chính:

Phương pháp nghiên cứu lý thuyết: Tổng hợp các công bố liên quan đến các mô hình, phương pháp chuyển đổi OpenMP sử dụng trên hệ thống máy tính với bộ nhớ phân tán. Phân tích, đánh giá ưu và khuyết điểm của các đề xuất đã công bố để làm cơ sở cho việc cải tiến và/hoặc đề xuất mới.

Phương pháp mô phỏng, thực nghiệm: Phát triển hệ thống phần mềm tương ứng với mô hình đề xuất mới. Thực nghiệm việc cài đặt, chạy thử phần mềm đã phát triển được trên các bài toán mẫu, thu thập và phân tích kết quả nhằm nhắm đánh giá tính đúng đắn và hiệu năng tính toán của mô hình đề xuất.

Cấu trúc luận án

Luận án bao gồm phần mở đầu, bốn chương nội dung, phần kết luận và danh mục các tài liệu tham khảo. Cụ thể:

Chương 1 "**Tổng quan nghiên cứu**" giới thiệu tổng quan các lý thuyết, khái liên quan đến nội dung nghiên cứu; tổng quan tài liệu hướng nghiên cứu chuyển đổi OpenMP lên hệ thống bộ nhớ phân tán từ đó chỉ ra được các nhiệm vụ cần giải quyết của luận án.

Chương 2 **"CAPE trên hệ thống tính toán đa lõi"** đây chính là phần trọng tâm của luận án, trình bày đề xuất mô hình CAPE mới - **CAPE song song hai mức đa luồng (gọi tắt là CAPE đa luồng)** - và kết quả thực nghiệm sử dụng CAPE mới khai thác được hiệu năng của CPU đa lõi.

Chương 3 "**Kỹ thuật chụp ảnh tiến trình cho CAPE đa luồng** " trình bày yếu tố kỹ thuật cốt lõi của CAPE, phát triển kỹ thuật chụp ảnh tiến trình áp dụng cho CAPE đa luồng.

Chương 4 "**Xử lý các vấn đề chia sẻ dữ liệu của CAPE đa luồng**" trình bày việc chia sẻ và đồng bộ dữ liệu của CAPE đa luồng.

Phần "**Kết luận và hướng phát triển của luận án**" nêu những đóng góp của luận án và hướng phát triển trong tương lai.

CHƯƠNG 1 TỔNG QUAN NGHIÊN CỨU

Chương này sẽ giới thiệu tổng quan về các lý thuyết, khái niệm liên quan đến đề tài nghiên cứu và các công trình nghiên cứu liên quan về việc cài đặt API OpenMP lên các hệ thống sử dụng bộ nhớ phân tán, hướng nghiên cứu mà luận án đang theo đuổi.

1.1 Tính toán hiệu năng cao

Tính toán Hiệu năng cao (High performance computing (HPC)) đề cập đến việc tổng hợp sức mạnh tính toán theo cách mang lại hiệu suất cao hơn nhiều so với máy tính thông thường hoặc máy trạm (workstation) để giải quyết các vấn đề lớn trong khoa học, kỹ thuật hoặc kinh doanh.

HPC thường đề cập đến việc xử lý các phép tính phức tạp ở tốc độ cao trên nhiều máy chủ song song. Các nhóm máy chủ đó được gọi là cụm và bao gồm hàng trăm hoặc thậm chí hàng nghìn máy chủ máy tính đã được kết nối thông qua một mạng và thường được gọi là siêu máy tính. Trong một cụm HPC (HPC cluster), mỗi máy tính thành phần thường được gọi là một nút.

Các cụm HPC chạy hàng loạt tính toán. Cốt lõi của bất kỳ cụm HPC nào là bộ lập lịch, được sử dụng để theo dõi các tài nguyên có sẵn, cho phép các yêu cầu công việc được gán một cách hiệu quả cho các tài nguyên máy tính khác nhau (CPU và GPU) thông qua mạng nhanh.

Một giải pháp HPC điển hình có 3 thành phần chính:

+ Tính toán
+ Mạng
+ Lưu trữ dữ liệu

Các giải pháp HPC có thể được triển khai tại chỗ hoặc trên “đám mây”.

1.2 Tính toán song song

Tính toán song song (Parallel Computing) hay xử lý song song (Parallel Processing): quá trình xử lý thông tin trong đó nhấn mạnh việc nhiều đơn vị dữ liệu được xử lý đồng thời bởi một hay nhiều bộ xử lý để giải quyết một bài toán.

1.2.1 Các hình thức tổ chức

Máy tính song song dựa trên bộ vi xử lý: được thiết kế với rất nhiều bộ xử lý có tốc độ vừa phải.

Song song về dữ liệu (data parallelism): là cơ chế sử dụng nhiều đơn vị xử lý thực hiện cùng một thao tác trên nhiều đơn vị dữ liệu.

Song song điều khiển (control parallelism) : là cơ chế trong đó nhiều thao tác khác nhau tác động lên nhiều đơn vị dữ liệu khác nhau một cách đồng thời.

Dây chuyền (pipeline) : là cơ chế chia công việc thành nhiều chặng nối tiếp, mỗi chặng được thực hiện bởi một bộ phận khác nhau. Đầu ra của bộ phận này là đầu vào của bộ phận tiếp theo.

Hệ số Tăng tốc (speedup) : hệ số tăng tốc của chương trình song song là tỉ số giữa thời gian thực hiện trong tình huống sử dụng chương trình tuần tự và thời gian thực hiện cũng công việc đó của chương trình song song.

$$Hệ\ số\ tăng\ tốc = \frac{t_{tt}}{t_{ss}}$$

Trong đó:

t_{tt} : thời gian thực hiện trong tình huống xấu nhất của chương trình tuần tự nhanh nhất;

t_{ss} : thời gian thực hiện trong tình huống xấu nhất của chương trình song song đang xét.

1.2.2 Các mức độ song song

Giả sử có 10 công việc từng đôi một khác nhau được giao cho 10 máy thực hiện, ta có mức song song cao nhất, và chúng ta gọi là mức chương trình song song. Mỗi công việc ta lại chia thành các công đoạn (Task) và có thể thực hiện song song, ta gọi mức độ song song này là mức song song chương trình con. Mỗi chương trình cũng như chương trình con lại có hàng loạt câu lệnh, ta có mức độ song song câu lệnh, trong câu lệnh lại có hành loạt thao tác (operation) => mức độ song song thao tác.

Theo luật Amdahl [36], dự đoán về hệ số tăng tốc độ của chương trình xử lý song song được tính theo công thức sau:

$$S_{\text{latency}} = \frac{1}{(1-p) + \frac{p}{s}}$$

Trong đó

$S_{latency}$: hệ số tăng tốc lý thuyết toàn cục

p: tỉ lệ có thể song song hóa của thuật toán tuần tự

s: số bộ xử lý song song.

Thực tế với chương trình song song, hiệu năng còn ảnh hướng từ rất nhiều yếu tố: chi phí thời gian để chia công việc, gửi công việc và nhận/tổng hợp kết quả từ các máy trạm, ảnh hưởng của phần cứng máy tính khi chạy nhiều chương trình đồng thời. Cũng có một số giải thuật thực hiện hiệu quả khi chạy tuần từ trên 1 CPU nhưng chạy không đạt hiệu quả cao khi chạy song song trên nhiều CPU vì đơn giản việc chạy tuần tự không mất thời gian phối hợp giữa các luồng chạy song song và cũng có thể thuật toán trong CPU khi chạy đơn lõi lại phù hợp hơn với phần cứng bên dưới hoạt động (CPU pipelines, CPU cache ,…).

1.2.3 Phân loại các kiến trúc song song

Một trong những phân loại hay được nhắc tới của Flynn – 1972. Michael Flynn phân các kiến trúc máy tính thành bốn loại dựa trên tương tác giữa lệnh và dữ liệu :

- SISD (Single Instruction stream, Single Data stream): Kiến trúc tuần tự Von Neuman, trong đó tại mỗi thời điểm chỉ một lệnh được thực hiện.

- MISD (Multiple Instruction stream, Single Data stream): Kiến trúc này cho phép một vài lệnh cùng thao tác trên một dữ liệu

- SIMD (Single Instruction stream, Multiple Data stream): Cho phép một lệnh được thực hiện đồng thời trên các dữ liệu khác nhau.

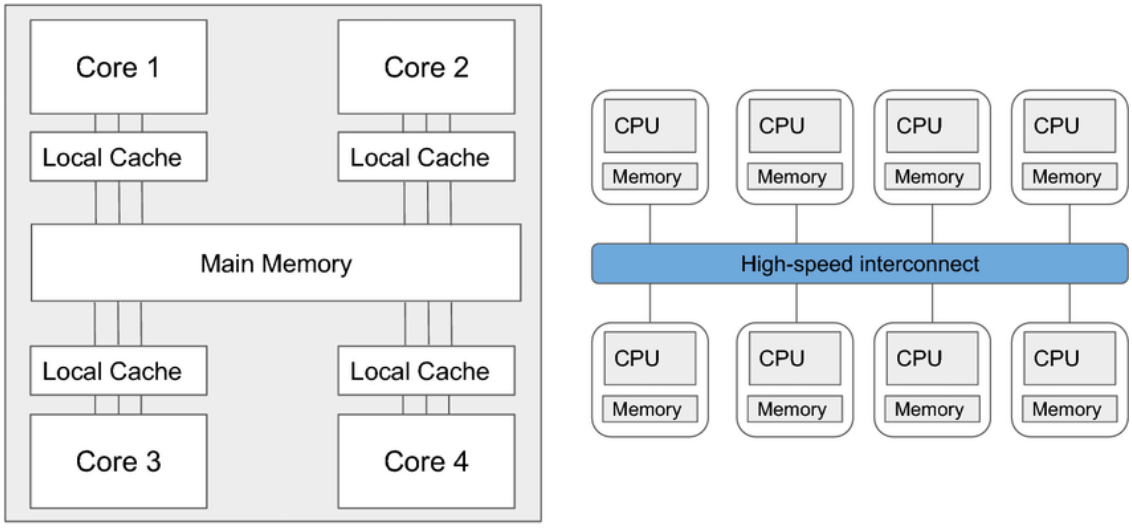
- MIMD (Multiple Instruction stream, Multiple Data stream): Cho phép nhiều lệnh khác nhau có thể đồng thời xử lý nhiều dữ liệu khác nhau trong cùng một thời điểm.

1.3 Máy tính đa CPU, CPU đa lõi (core) và đa luồng

Hệ thống máy tính đa CPU (multi-CPU systems) và CPU đa lõi (multicore systems) được đề cấp tiếp theo đều thuộc kiến trúc MIMD.

Trước khi có CPU đa lõi để tăng năng lực tính toàn con người đã cố gắng thêm sức mạnh xử lý cho máy tính bằng cách bổ sung thêm CPU trên một hệ thống máy tính được gọi là hệ thống máy tính đa CPU. Điều này đòi hỏi một bo mạch chủ có nhiều ổ cắm CPU (CPU sockets). Bo mạch chủ cũng cần bổ sung thêm phần cứng để kết nối các CPU sockets đó với RAM và các tài nguyên khác.

Các hệ thống máy đa CPU có thể được chia thành hai loại theo các cách tổ bộ nhớ khác nhau, một loại sử dụng hệ thống bộ nhớ chia sẻ (như minh họa ở Hình 1.1a) và loại kia sử dụng hệ thống bộ nhớ phân tán (như mình họa ở Hình 1.1b).

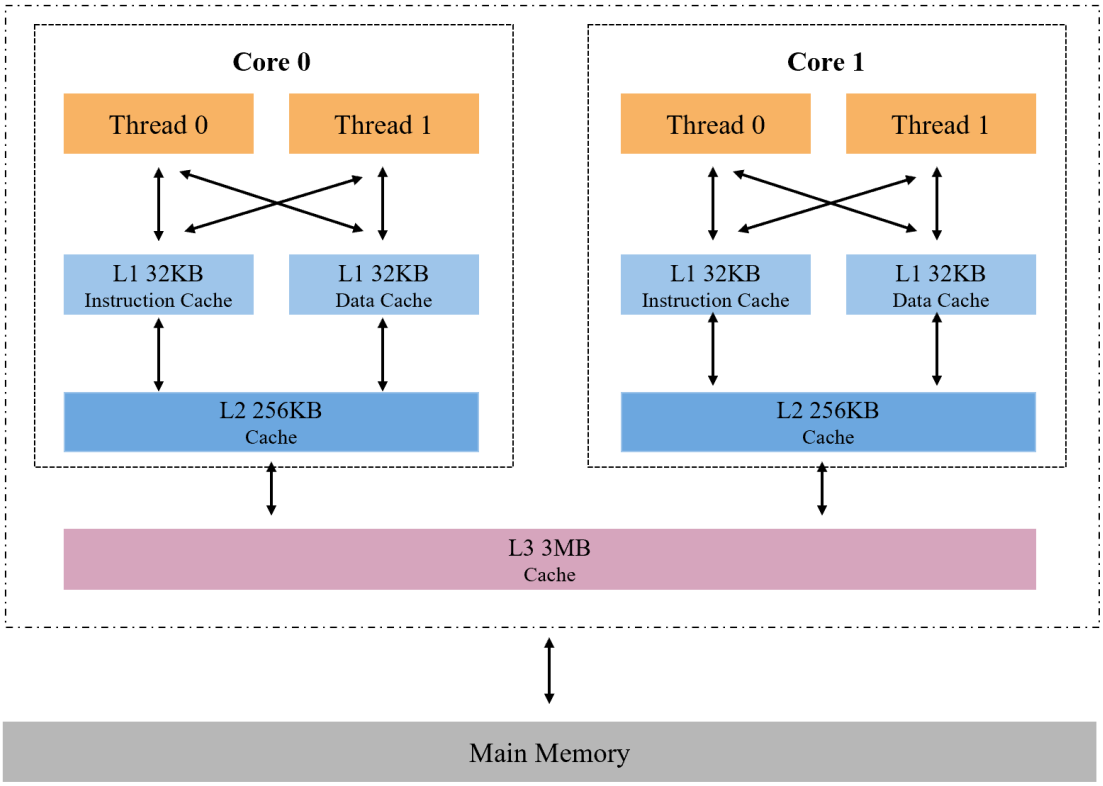


Hình 1.1. Minh họa hệ thống máy đa CPU

Hệ thống máy tính CPU đa lõi xử dụng CPU đã lõi, có nhiều đơn vị vị xử lý trung tâm (liều lõi) nhưng chúng vẫn nằm gọn trong một khe cắm CPU trên bo mạch chủ, tất cả đều sử dụng cùng nguồn điện, khả năng làm mát và phần cứng khác. Hình 1.2 minh họa hệ thống máy tính CPU đa lõi, gồm có 2 lõi, mỗi lõi sử dụng công công nghệ siêu phân luồng (hyper-threading) [75].

Công nghệ siêu phân luồng là một cải tiến phần cứng cho phép nhiều luồng cùng chạy trên mỗi lõi. Nhiều luồng hơn có nghĩa là có nhiều công việc được thực hiện song song hơn. Khi Công nghệ siêu phân luồng hoạt động, CPU sẽ hiển thị hai bối cảnh thực thi trên mỗi lõi vật lý. Điều này có nghĩa một lõi vật lý hiện hoạt động giống như hai "lõi logic" có thể xử lý các luồng phần mềm khác nhau.

Hai lõi logic có thể xử lý các tác vụ hiệu quả hơn so với lõi đơn luồng truyền thống. Bằng cách tận dụng thời gian không hoạt động mà trước đây lõi chờ các tác vụ khác hoàn thành, Công nghệ siêu phân luồng sẽ cải thiện hiệu suất xử lý CPU lên tới 30% trong các ứng dụng máy chủ [75].



Hình 1.2. Minh họa hệ thống máy CPU đa lõi

Với minh họa ở Hình 1.2, CPU có 2 lõi (core) áp dụng công nghệ siêu phân luồng tạo thành 4 luồng. Bộ nhớ đệm (cache) trong CPU đảm bảo cho các luồng có thể truy cập dữ liệu với tốc độ cao. Có hai loại bộ đệm L1: L1i - lưu trữ chỉ dẫn điều khiển và L1d - lưu trữ dữ liệu. Các luồng khác nhau trong cùng một lõi chia sẻ bộ đệm L1 và L2 và các lõi khác nhau giao tiếp thông qua bộ đệm L3. Cùng một luồng chỉ có thể chạy trên một lõi CPU tại một thời điểm.

CPU và lõi là yếu tố phần cứng, hệ điều hành là yếu tố phần mềm hệ thống giúp các ứng dụng tầng trên giao tiếp với phần cứng. Các tiến trình/chương trình (process) khác nhau chạy trên hệ điều hành được cấp phát vùng nhớ để chạy chương trình tách biệt nhau. Một tiến trình sở hữu một hoặc nhiều luồng (thread). Luồng có thể được xem là tiến trình con nằm trong tiến trình. Số luồng trong một tiến trình có thể không cố định. Tiến trình có thể huỷ hoặc sinh ra luồng mới khi cần thiết. Các luồng nằm trong cùng một tiến trình và điều này cho phép các luồng đọc và viết lên cùng một không gian bộ nhớ chung, do đó các luồng dễ dàng có thể giao tiếp dễ dàng với nhau. CPU điều khiển các lõi thực hiện các câu lệnh trong luồng. Tại mỗi thời điểm, một lõi chỉ làm việc với một luồng và chỉ xử lý một nhiệm vụ (task) của luồng đó.

Đa luồng (Multithreading) là khả năng xử lý nhiều luồng cùng lúc của CPU. Khi có nhiều nhiệm vụ khác nhau, CPU có thể làm việc một cách song song (parallel) nhiều luồng đối với CPU chỉ có một lõi hoặc làm việc đồng thời (concurrent) đối với CPU đa lõi. Lõi là nơi thực hiện các nhiệm vụ và mỗi lõi chỉ chạy 1 nhiệm vụ tại một thời điểm. Chính vì lí do này mà có càng nhiều lõi, CPU càng thực hiện được nhiều nhiệm vụ cùng lúc. Như đã trình bày ở đoạn trên, Công nghệ siêu phân luồng của Intel sẽ cải thiện hiệu suất xử lý CPU lên tới 30% [75].

Hầu hết các phiên bản hệ điều hành phổ biến hiện nay như Windows, phiên bản phân phối của Linux đều hỗ trợ cho CPU đa lõi cũng như đa luồng.

Mục tiêu và phạm vi nghiên cứu cải tiến CAPE của luận án này hướng đến áp dụng cho hệ thống máy tính sử dụng CPU đa lõi. Cần lưu ý rằng CAPE, cũng tương tự như MPI, ngôn ngữ trừu tượng bậc cao, bao gồm các thư viện runtime ở tầng ứng dụng và không can thiệp điều khiển trực tiếp phần cứng của CPU đa lõi mà sử dụng lại các dịch vụ của hệ điều hành cung cấp để chạy chương trình. Với minh họa như ở Hình 1.2, hệ điều hành hỗ trợ CPU đa lõi sẽ cung cho các ứng dụng và ngôn ngữ lập trình tầng trên của máy tính có CPU 4 lõi, việc lập lịch điều phối xử lý trực tiếp ở CPU đa lõi do hệ điều hành quản lý.

1.4 OpenMP

1.4.1 Giới thiệu về OpenMP

Như đã được giới thiệu ở **Phần Mở đầu** mục **Tính cấp thiết của đề tài**, OpenMP là một giao diện lập trình (API) cung cấp một mức trừu tượng hóa cao để viết các chương trình song song cho các hệ thống tính toán hiệu năng cao với ưu điểm dễ học, dễ sử dụng và ít tốn công sức lập trình khi so sánh với thư viện lập trình khác như MPI, điều này được minh hoạ rõ trong ví dụ về chương trình nhân 2 ma trận dưới đây, với mã nguồn đầu tiên được viết bằng C và OpenMP; mã nguồn thứ 2 bằng C và MPI.

```
#include <stdio.h>
#include <omp.h>
#define N 100000
int A[N][N], B[N][N], C[N][N];
int main(){
    int i, j, k;
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++){
            A[i][j] = rand()%10;
            B[i][j] = rand()%10;
```

```
            C[i][j] = 0;
    }
    #pragma omp parallel for private(j, k)
    for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
            for(k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];
    return 0;
}
```

Hình 1.3 Chương trình nhân hai ma trận viết bằng OpenMP

Ở Hình 1.3, chương trình OpenMP, dễ dàng nhận thấy đây là một chương trình C bình thường (tuần tự), chỉ được bổ sung thêm 2 dòng liên quan đến OpenMP: 1) dòng `#include <omp.h>` ở đầu chương trình, có chức năng khai báo việc sử dụng thư viện OpenMP; và 2) dòng chỉ thị `#pragma omp parallel for private(j, k)`, có chức năng chỉ định vòng lặp `for` tiếp sau đó sẽ được xử lý một cách song song, với những thông số song song mặc định của OpenMP và hai biến `j, k` biến cục bộ trong mỗi luồng. Tất cả những công việc khác để thiết lập môi trường thực hiện song song như tạo các luồng, phân phối công việc vào các luồng, đồng bộ hoá các luồng,… đều được thực hiện bởi trình biên dịch và thư viện OpenMP.

Dưới đây chương trình nhân 2 ma trận như trên, được viết bằng MPI.

```
#include <stdio.h>
#include "mpi.h"
#define N 100000
MPI_Status status;
int A[N][N], B[N][N], C[N][N];
int main(){
    int processCount, processId, slaveTaskCount, source, dest, rows,offset;
    struct timeval start, stop;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &processId);
    MPI_Comm_size(MPI_COMM_WORLD, &processCount);
    slaveTaskCount = processCount - 1;
    if (processId == 0){ //Master process
        for (int i = 0; i<N; i++){
            for (int j = 0; j<N; j++){
                A[i][j]= rand()%10;
                B[i][j]= rand()%10;
            }
        }
    }
    rows = N/slaveTaskCount;
    offset = 0;
    for (dest=1; dest <= slaveTaskCount; dest++){
        MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
```

```
        MPI_Send(&rows, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
        MPI_Send(&A[offset][0], rows*N, MPI_DOUBLE,dest,1,
               MPI_COMM_WORLD);
        MPI_Send(&B, N*N, MPI_DOUBLE, dest, 1, MPI_COMM_WORLD);
        offset = offset + rows;
    }
    for (int i = 1; i <= slaveTaskCount; i++){
        source = i;
        MPI_Recv(&offset, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, source, 2, MPI_COMM_WORLD, &status);
        MPI_Recv(&C[offset][0], rows*N, MPI_DOUBLE, source, 2,
                MPI_COMM_WORLD, &status);
    }
    if (processId > 0) {  // Slave Processes
        source = 0;
        MPI_Recv(&offset, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&rows, 1, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&A, rows*N, MPI_DOUBLE, source, 1, MPI_COMM_WORLD,
             &status);
        MPI_Recv(&B, N*N, MPI_DOUBLE, source, 1, MPI_COMM_WORLD,
             &status);
        for (int k = 0; k<N; k++){
            for (int i = 0; i<rows; i++){
                C[i][k] = 0.0;
                for (int j = 0; j<N; j++)
                     C[i][k] = C[i][k]+A[i][j] * B[j][k];
            }
        }
        MPI_Send(&offset, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
        MPI_Send(&rows, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
        MPI_Send(&C, rows*N, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Hình 1.4. Chương trình nhân 2 ma trận viết bằng MPI

Hình 1.4 trình bày chương trình viết bằng MPI. Chương trình này đòi hỏi người lập trình phải xác lập một cách tường minh tất cả các vấn đề bao gồm cấu trúc của hệ thống (bao gồm các tiến trình chính và phụ), tổ chức phân chia công việc, gửi nhận dữ liệu, đồng bộ hoá các tiến trình,…

Rõ ràng, MPI đòi hỏi ở lập trình viên khả năng, công sức lập trình nhiều hơn nhiều lần so với sử dụng OpenMP. Vì thế, dù MPI có khả năng cung cấp hiệu năng cao nhất trên các hệ thống máy tính bộ nhớ phân tán, việc đưa OpenMP lên các hệ thống máy tính bộ nhớ phân tán là một vấn đề rất có ý nghĩa thực tế trong lĩnh vực phát triển phần mềm.

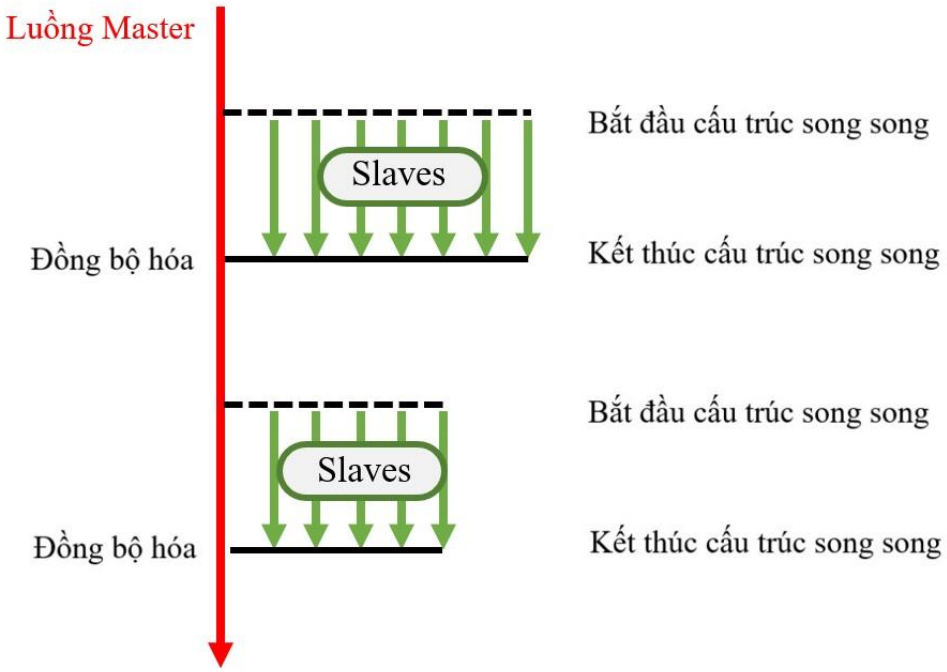
1.4.2 Mô hình hoạt động của OpenMP

OpenMP sử dụng mô hình *fork-join* của quá trình thực thi song song. Nhiều luồng (thread) thực thi thực hiện các tác vụ được định nghĩa ngầm hoặc tường minh bởi các chỉ thị OpenMP. OpenMP có mục tiêu hỗ trợ các chương trình sẽ thực thi chính xác cả dưới dạng chương trình song song (nhiều luồng thực thi và một thư viện hỗ trợ OpenMP đầy đủ) như các chương trình tuần tự.

Một chương trình OpenMP bắt đầu như một chương trình đơn luồng, luồng này được gọi là một luồng ban đầu. Một luồng ban đầu thực thi tuần tự, với các vùng song song tiềm ẩn trong toàn bộ chương trình.

Khi một luồng của chương trình gặp một cấu trúc song song, sẽ tạo một nhóm (team) của chính nó có thể gồm không hoặc nhiều luồng bổ sung và trở thành luồng chính (master) của nhóm mới. Một tập hợp các nhiệm vụ ngầm định, một nhiệm vụ cho mỗi luồng, được tạo ra. Mã cho mỗi tác vụ được xác định bởi mã bên trong cấu trúc song song. Mỗi nhiệm vụ được giao cho một luồng khác nhau trong nhóm và có ràng buộc quản lý bởi luồng mà nó được gán ban đầu. Vùng nhiệm vụ của tác vụ đang được thực thi bởi một luồng trong nhóm gặp phải tình huống bị tạm ngưng thì các luồng khác trong nhóm vẫn có thể tiếp tục thực hiện nhiệm vụ ngầm định của chúng. Một rào chắn (barrier) ngầm được đặt ở cuối vùng song song để đồng bộ hóa việc hoàn thành công việc của các luồng nếu không có điều khoản *nowait (không cần đợi)* được chỉ định. Chỉ luồng chính tiếp tục thực thi sau phần cuối của cấu trúc song song, tiếp tục vùng tác vụ đã bị treo khi gặp cấu trúc song song. Bất kỳ số lượng cấu trúc song song nào đều có thể được chỉ định trong một chương trình. Các vùng song song có thể được lồng vào nhau một cách tùy ý.

Mô hình hoạt động này được minh họa trong Hình 1.5.



Hình 1.5. Mô hình hoạt động của OpenMP

Nhiều trình biên dịch OpenMP đã được phát triển cho Linux, Windows, MacOS, Solaris, FreeBSDTHER [3]. Tuy nhiên, liên quan đến mô hình đa luồng và mô hình bộ nhớ chia sẻ, OpenMP không thể chạy trên các hệ thống sử dụng bộ nhớ phân tán. Do đó, việc làm sao để có thể đưa được OpenMP lên trên các hệ thống sử dụng bộ nhớ phân tán trở thành một chủ đề được nhiều nhóm nghiên cứu thực hiện. Luận án này cũng nằm trong phạm vi của vấn đề trên, vì vậy trước hết, phải tiến hành hệ thống hóa và khái quát hóa các nghiên cứu đã và đang thực liên quan đến hướng nghiên cứu của đề tài.

1.5 Các công trình nổi bật về chuyển đổi OpenMP lên hệ thống bộ nhớ phân tán

1.5.1 Phương pháp sử dụng SSI làm bộ nhớ chung cho tất cả các thread

SSI (Single System Image) là một hệ thống các máy tính được liên kết với nhau (ảo hóa) tạo thành một máy thống nhất với bộ nhớ chung. SSI cung cấp ở mức trừu tượng một bộ nhớ chia sẻ trên nền kiến trúc bộ nhớ phân tán vật lý. Có nhiều giải pháp để thực hiện tạo ra hệ thống SSI như Kerrighed [21] và OpenSSI [22]. Cài đặt OpenMP trên kiến trúc SSI được tiếp cận mang tính trực quan nhất và có khả năng tương thích hoàn toàn với chuẩn OpenMP. Tất cả các yếu tố của OpenMP như mô hình thực hiện, phương thức quản lý bộ nhớ và cơ chế đồng bộ đều có thể dễ dạng cài đặt được. Tuy nhiên nhược điểm chính của phương pháp này là hiệu suất không cao. Việc truy xuất mở rộng bộ nhớ trên máy tính khác có một một độ trễ nhất định, đồng thời cũng yêu cầu

cơ chế đồng bộ nữa. Cả hai yếu tố này làm ảnh hưởng chung đến hiệu suất của toàn chương trình. Do đó, các nghiên cứu và phát triển trên Kerrighed và OpenSSI không còn được tập trung nghiên cứu, và cho đến nay phiên bản cập nhật cuối cùng của Kerrighed và OpenSSI lần lượt vào các năm 2010 và 2013.

1.5.2 Phương pháp ánh xạ một phần không gian nhớ của luồng

Để giảm bớt ảnh hưởng của việc ánh xạ toàn bộ không gian nhớ của các luồng trên một vùng nhớ ảo chia sẻ chung, cách tiếp cận này chỉ ánh xạ một phần của vùng nhớ cần thiết trên vùng nhớ chia sẻ (Distributed Shared Memory - DSM). Theo đó, việc đồng bộ dữ liệu chỉ thực hiện trên một phần vùng nhớ chia sẻ và nhờ vậy giảm thiểu được thời gian truy cập lên các vùng nhớ còn lại. Vấn đề còn lại phải làm thế nào để chỉ định được các biến chia sẻ để ánh xạ lên vùng nhớ chia sẻ. Đã có một số công trình nghiên cứu để giải quyết vấn đề này.

Cách tiếp cận đầu tiên, mở rộng thêm API của OpenMP, chẳng hạn như công cụ Cluster OpenMP của Intel [11] đã thêm các chỉ thị mới để chỉ định việc chia sẻ dữ liệu. Các chỉ thị mới này sẽ được các luồng sử dụng trong việc chia sẻ dữ liệu. Một số biến được chia sẻ tự động bởi trình biên dịch (compiler) như việc cấp phát stack và truyền các tham trị chuẩn của các hàm con. Một số biến cần phải được lập trình viên chỉ định cụ thể tường minh, ví dụ như file-scope (như là biến toàn cục) trong C và C++. Những điều này lại dẫn đến việc Cluster OpenMP không còn tương thích hoàn toàn với API OpenMP gốc. Hiện tại Intel cũng đã dừng phát triển Cluster OpenMP.

Cách tiếp cận khác, sử dụng mô hình SCASH [7], mô hình này được để xuất bởi Mitsuhisa Sato và các cộng sự trong đó sử dụng một trình kết hợp với trình biên dịch chuyển đổi mã Omni [22] để chuyển chương trình OpenMP chuyển sang chương trình mã C. Bản chất của SCASH là thư viện lệnh cho phép thực hiện truy cập bộ nhớ từ xa như lệnh đọc ("`shmem_get`") hoặc lệnh ghi ("`shmem_put`"), với yêu cầu thư viện SHMEM được cài đặt sẵn trên các máy tính sử dụng đã được kết nối mạng với nhau. Trong mô hình này, không gian địa chỉ dùng chung phải được khai báo tường minh và được cấp phát đầu tiên khi chạy chương trình. Để biên dịch một chương trình OpenMP sang mô hình SCASH, chương trình dịch phải biên dịch mã cấp phát bộ nhớ toàn cục sang vùng nhớ chia sẻ tại thời điểm chạy chương trình. Mô hình SCASH có thể tạo được hiệu quả cao cho chương trình OpenMP nếu dữ liệu cần thiết của mỗi luồng được tổ

chức trên bộ vi xử lý của luồng đang chạy (trên cùng máy vật lý). Để tạo ra được môi trường như vậy, Omni mở rộng tập chỉ thị của OpenMP để cho phép lập trình viên có thể chỉ định không gian địa chỉ vùng nhớ chia sẻ và cần bổ sung thêm chỉ thị lặp đồng bộ để lập lịch đồng bộ dữ liệu của các thread cần chia sẻ dữ liệu với nhau. Tương tự với Cluster OpenMP, việc thêm các chỉ thị riêng của mình đã làm Omni không còn tương thích hoàn toàn với API OpenMP nguyên gốc nữa.

Để nâng hiệu năng theo phương pháp SCASH, Mitsuhisa Sato và các cộng sự xây dựng trình biên dịch chuyển đổi mã XcalableMP [23] sử dụng mảng phối hợp để trao đổi dữ liệu giữa các nút. Bản chất việc trao đổi dữ liệu giữa các nút sử dụng thư viện MPI với các hàm `MPI_Get()`, `MPI_Put()`,`MPI_Accumulate()` và `MPI Fence()`. Kết quả đánh giá cho thấy hiệu năng cao tuy nhiên cần bổ sung thêm các chỉ thị riêng. Hiện này phương pháp này đang tiếp tục phát triển theo hướng bổ sung thêm nhiều chỉ thị riêng đồng thời có thể phối hợp với MPI. Phiên bản XcalableMP đang cập nhật đến phiên bản 1.4 xuất bản năm 2018.

1.5.3 Phương pháp sử dụng mô hình HLRC

Phương pháp này sử dụng mô hình HLRC (Home-based Lazy Release Consistency) [24] để cài đặt mô hình đồng bộ trễ (Relaxed Consistency) của OpenMP. Mô hình HLRC sử dụng cơ chế đảm bảo đồng bộ đồng nhất giữa các nút. Bằng cách sử dụng cơ chế trang nhớ ảo để phát hiện các truy cập đến bộ nhớ chia sẻ và những nội dung cần được cập nhật trên các vùng nhớ ảo với sự hỗ trợ của giải pháp thiết bị phần cứng InfiniBand [25]. Cụ thể, mỗi trang bộ nhớ trong hệ thống HLRC luôn có một bản copy dữ liệu cập nhật nhất được lưu giữ trên nút chính. Khi một xử lý cần truy cập đến một trang dữ liệu cục bộ của một nút khác chưa được cập nhật mới nhất lập tức một bản dữ liệu sao chép từ nút chính sẽ được sử dụng để cập nhật ngay lập tức trang nhớ đó trên nút này. Nhiều nút có quyền ghi vào một trang với cấu trúc tổ chức dữ liệu cho phép kiểm soát được sự thay đổi dữ liệu của trang. Các thay đổi đều được đồng bộ đến nút chính.

Mặc dù việc sử dụng mô hình HLRC có thể giảm việc thực hiện truy cập đến vùng bộ nhớ chia sẻ đem lại hiệu suất cao nhưng mô hình này cũng gặp một số vấn đề khác. Đầu tiên là việc khó để chỉ định tự động tất cả các biến chia sẻ vì OpenMP không yêu cầu khai báo tường minh tất cả các biến chia sẻ. Thứ hai, làm thế nào để việc thực hiện

chỉ thị đồng bộ toàn bộ không nhớ `flush` và khởi tạo các biến chia sẻ với tốc độ cao nếu không có sự hỗ trợ của phần cứng chuyên dụng. Đồng thời, việc giảm các mệnh đề và tối thiểu hóa các chỉ thị tạo ra sự phức tạp hơn khi cài đặt OpenMP theo mô hình này.

1.5.4 Phương pháp kết hợp với MPI

Có một số nghiên cứu thực hiện dịch mã OpenMP sang mã MPI với mục đính kết hợp ưu điểm của cả OpenMP (tính đơn giản, dễ học, dễ sử dụng) và MPI (hiệu năng cao) trên hệ thống máy tính có bộ nhớ phân tán.

MPI sử dụng cơ chế bộ nhớ phân tán theo đó mỗi nút của hệ thống có một không gian bộ nhớ riêng, không được chia sẻ với hệ thống khác. Trong chương trình song song viết theo MPI, lập trình viên phải chỉ định tường minh việc chia sẻ dữ liệu giữa các nút, trong khi đối với OpenMP tất cả các biến được chia sẻ được thực hiện ngầm định.

Để giải quyết sự khác nhau này, các ý tưởng thực hiện theo phương pháp cài đặt này biên dịch chương OpenMP ra chương trình MPI [8][26] hoặc biên dịch ra chương trình trình chạy C++ hoặc Fortran [9] bổ sung các chỉ thị giao tiếp của MPI.

Cả [8][8][26] đều sử dụng các chiến lược và thuật toán khác nhau để phân tích tự động dữ liệu cần chia sẻ cũng như giải quyết mối quan hệ nơi gửi và nơi nhận. Các cài đặt theo phương pháp này có đánh giá bước đầu cho kết quả hiệu năng cao. Tuy nhiên, tất cả các cài đặt này đều chưa thể cài đặt được một cách tự động toàn bộ các chỉ thị của OpenMP.

1.5.5 Phương pháp dựa trên Mảng toàn cục (Global Array - GA)

Một cài đặt OpenMP trên hệ thống máy tính sử dụng bộ nhớ phân tán bằng cách sử dụng GA đã được L.Huang và các cộng sự giới thiệu [10][28][29]. GA một thư viện được thiết kế để dùng cho tính toán song song trên hệ thống có bộ nhớ phân tán. GA đơn giản hoá lập trình song song bằng cách cung cấp cho người dùng khái niệm của bộ nhớ chia sẻ ảo cho các hệ thống bộ nhớ phân tán. Người lập trình có thể viết chương trình song song trên hệ thống những hệ thống cluster, nếu cần sử dụng biến chia sẻ trong chương trình, chỉ cần chia sẻ dữ liệu tầng trên mà không quan tâm tổ chức ở mức vật lý bên dưới. Những chương trình GA phân bố dữ liệu theo nhiều khối để phân cho các tiến trình. Trước khi một đoạn mã được thực hiện, các dữ liệu cần thiết phải được thu thập từ các tiến trình xử lý liên quan; các dữ liệu này được phân phối về lại các vị trí vật lý của các tiến trình. GA thực hiện truyền dữ liệu dựa trên MPI. L. Huang và các cộng sự

đã thực hiện hầu hết các chỉ thị của OpenMP, thư viện thường dùng và môi trường biến được phiên dịch sang thư viện GA hoặc MPI ở mức mã nguồn, ngoại trừ một số thiết lập động hoặc thay đổi số lượng thread, như các chỉ thị `omp_set_dynamic`, `omp_set_num_threads`. Ý tưởng chung của phương pháp là biên dịch OpenMP sang GA để khai báo tất cả các biến chia sẻ của chương trình OpenMP vào mảng toàn cục của GA.

Kết quả thực nghiệm của mô hình này được mô tả trong [10][28][29] đem lại hiệu suất cao. Tuy nhiên, hạn chế chính của mô hình này là việc phải yêu cầu chỉ định tường minh biến chia sẻ, điều này làm ảnh hưởng đến ưu điểm nổi bật nhất của OpenMP là tính đơn giản, dễ học, không cần khai báo tường minh các biến chia sẻ.

1.5.6 Phương pháp libMPNode cải tiến trên phương pháp SSI

Như đã trình bày trong mục 1.5.1, phương pháp SSI có thể chuyển đổi hoàn toàn OpenMP lên hệ thống bộ nhớ phân tán. Tuy nhiên, nhược điểm của phương pháp SSI này là hiệu năng hoạt động của hệ thống khi cài đặt theo phương pháp này chưa cao. Phương pháp libMPNode [30] dựa trên nền tảng của hệ điều hành Popcorn Linux [31][32][33] đã hỗ trợ tính năng SSI để xây dựng thư viện libMPNode hỗ trợ chuyển đổi OpenMP chạy trên hệ thống bộ nhớ phân tán có thể chạy tối ưu hơn. libMPNode đã đề xuất cơ chế biên dịch chương trình OpenMP cũng như đồng bộ dữ liệu chia sẽ của chương trình OpenMP khi chạy trên hệ thống bộ nhớ phân tán trên nền tảng hệ điều hành SSI: Popcorn Linux. Ưu điểm của phương pháp này là có thể chuyển đổi hoàn chỉnh OpenMP chạy được trên hệ thống bộ nhớ phân tán, bước đầu đem lại hiệu năng hệ thống cao so với hệ thống trước đó sử dụng phương pháp SSI. Tuy nhiên, phương pháp này đang trong quá trình nghiên cứu chưa có thực nghiệm đánh giá so sánh hiệu năng với các hệ thống viết theo MPI (phương pháp cho hiệu năng cao nhất hiện nay).

1.5.7 Phương pháp OMPC sử dụng trình biên dịch chuyển đổi OpenMP chạy trên hệ thống bộ nhớ phân tán ứng dụng các thư viện MPI.

Phương pháp này đang có những cập nhật kết quả rất mới được công bố trong năm 2022, phương pháp OMPC [34] này có thể được phân loại vào nhóm phương pháp kết hợp với MPI như đã trình bày trong mục 1.5.4. Tuy nhiên, phương pháp này sử dụng trình biên dịch để chuyển đổi OpenMP chạy trên hệ thống bộ nhớ phân tán ứng dụng các thư viện của MPI tại thời điểm chạy (runtime) nên có thể xem việc sử dụng MPI là

trong suốt với người sử dụng. Người sử dụng có thể dễ dạng chạy OpenMP trên hệ thống bộ nhớ phân tán như chạy OpenMP trên hệ thống bộ nhớ chia sẽ. Phương pháp này có thể chuyển đổi hoàn chỉnh OpenMP lên hệ thống bộ nhớ phân tán. Tuy nhiên do hướng nghiên cứu này đang trong quá trình nghiên cứu nên các kết quả thực nghiệm ban đầu cho thấy hệ số tăng tốc của hệ thống khi sử dụng phương pháp này chưa ổn định và hiệu năng của hệ thống đang còn thấp hơn nhiều khi so sánh với hệ thống sử dụng MPI (đem lại hiệu năng hệ thống cao nhất hiện tại).

1.5.8 Phương pháp CAPE đơn luồng

Đây là phương pháp mà luận án tập trung vào nghiên cứu để cải tiến nên được trình bày chi tiết ở mục lớn tiếp theo, mục 1.7. Dưới sự hướng dẫn của GS. Éric Renault (Pháp), nhóm nghiên cứu trong nước gồm Hà Viết Hải, Nguyễn Cảnh Hoài Đức, Trần Văn Long và Đỗ Xuân Huyền đã liên tục nghiên cứu CAPE trong thời gian qua. Trong năm 2013, nhóm đã đăng một bài tại Hội thảo Quốc gia về Nghiên cứu và Ứng dụng Công nghệ thông tin - FAIR 2013 [20]. Bài báo chứng minh khả năng cung cấp hiệu năng cao của CAPE bằng cách so sánh mô hình hoạt động và các mô hình toán học tính toán hiệu năng của kỹ thuật này với MPI, công cụ đang được đánh giá có khả năng cung cấp hiệu năng cao nhất trên kiến trúc sử dụng bộ nhớ phân tán. Trong các năm từ 2015 đến 2018, nhóm nghiên cứu này đã công bố công trình ở các hội thảo khoa học và tạp chí uy tín về các cải tiến của CAPE về: (i) Cải tiến về cách truyền gửi dữ liệu giữa các nút tính toán từ việc sử dụng kỹ thuật socket sang sử dụng thư viện truyền dữ liệu của MPI [16] làm tăng độ tin cậy và ổn định trong việc truyền dữ liệu; (ii) Cải tiến tối ưu kích thước và thời gian thực hiện chụp ảnh tiến trình [17][18][19].

Từ năm 2016, nhóm bắt đầu tiến hành nghiên cứu để khai thác tốt hơn kiến trúc bộ vi xử lý đa lõi để gia tăng hiệu năng hoạt động của CAPE. Theo mô hình hoạt động hiện thời của CAPE, tại các vùng song song, mỗi nút phụ được giao một phần công việc của đoạn mã song song và thực hiện trên một tiến trình duy nhất. Chính điều này dẫn đến việc hầu như chỉ có một lõi CPU được sử dụng với tỷ lệ cao, còn các lõi khác ở trạng thái nghỉ. Để tăng được hiệu suất của chương trình, rõ ràng trong khoảng thời gian thực hiện tính toán ở các nút phụ, cần song song hóa các đoạn mã này, tức là phải tìm cách để có đa tiến trình (chương trình) hoặc đa luồng (thread) thực hiện. Mô hình hoạt động mới đã được chúng tôi đề xuất và thử nghiệm nhưng chưa thành công. Cụ thể, sử dụng

đa tiến trình trên mỗi nút phụ bằng cách cho thực hiện song song nhiều lần chương trình ứng dụng trên mỗi nút, mỗi lần thực hiện sẽ được chạy trong một tiến trình độc lập, có số hiệu khác nhau và được phân một phần công việc khác nhau của đoạn mã tính toán song song. Giải pháp này đã được cài đặt thử nghiệm, với cluster gồm các máy tính trang bị CPU Intel Core i3 (2 lõi x 2 siêu phân luồng), tốc độ 3.5GHz, RAM 4GB, chạy hệ điều hành Ubuntu 14.04, kết nối bằng mạng Ethernet tốc độ 100Mb/s (máy vật lý). Các thử nghiệm đã cho kết quả tốc tăng tốc được gần gấp đôi khi cho chạy 2 tiến trình ứng dụng đồng thời trên mỗi nút tính toán. Đây là một kết quả rất tốt và hợp lý về mặt hiệu năng. Tuy nhiên, giải pháp này gặp phải vấn đề về tính ổn định. Số liệu thực nghiệm cho thấy tình huống các tranh chấp về mặt tài nguyên và IP của hệ thống trên các nút tính toán dẫn đến dừng hệ thống lên đến đến gần 30% trường hợp thử nghiệm. Kết quả của hướng nghiên cứu này được công bố trong [64].

1.6 Tổng hợp đánh giá về các phương pháp chuyển đổi OpenMP trên kiến trúc bộ nhớ phân tán

Có nhiều phương pháp chuyển đổi OpenMP cho hệ thống máy tính sử dụng bộ nhớ phân tán. Nhìn chung, chưa có phương pháp nào thành công trên cả hai mặt tương thích hoàn toàn với API OpenMP và có hiệu năng cao. Cách tiếp cận SSI đã thành công khi cài đặt được toàn bộ các chỉ thị của OpenMP tuy nhiên hiệu năng chưa cao. Một số phương pháp khác đặt hiệu năng cao tuy nhiên chưa cài đặt xong các chỉ thị của OpenMP hoặc các bổ sung thêm chỉ thị riêng. Bảng 1.1 hệ thống lại đặc điểm của từng phương pháp.

Bảng 1.1. Tổng hợp so sánh các phương pháp cài đặt OpenMP trên kiến trúc bộ nhớ phân tán

Số TT	Phương pháp	Nguyên lý	Ưu điểm	Các tồn tại cần giải quyết
1	SSI	Xây dựng bộ nhớ ảo toàn cục ánh xạ đến bộ nhớ vật lý phân tán	Cài đặt đầy đủ chỉ thị của OpenMP	Hiệu năng thấp
2	Cluster OpenMP	Chỉ ánh xạ các biến chia sẻ vào vùng nhớ chia sẻ chung	Hiệu năng cao	Sử dụng các chỉ thị riêng, khác với OpenMP

Số TT	Phương pháp	Nguyên lý	Ưu điểm	Các tồn tại cần giải quyết
3	SCASH, XcalableMP	Chỉ ánh xạ các biến chia sẻ vào vùng nhớ chia sẻ chung	Hiệu năng cao	Sử dụng các chỉ thị riêng khác với OpenMP
4	HLRC	Sử dụng cơ chế trang nhớ ảo để phát hiện các truy cập đến bộ nhớ chia sẻ	Hiệu năng cao	Cần hỗ trợ của giải pháp phần cứng chuyên dụng
5	Kết hợp với MPI	Dịch các chỉ thị OpenMP sang MPI	Hiệu năng cao	Chưa cài đặt được nhiều chỉ thị của OpenMP
6	GA	Sử dụng mảng toàn cục	Hiệu năng cao	Chưa cài đặt được việc tự động chỉ định biến chia sẻ
7	libMPNode	Cải tiến dựa trên phương pháp SSI	Cài đặt đầy đủ chỉ thị của OpenMP	Dựa trên hỗ trợ hệ điều hành riêng. Chưa có nhiều thực nghiệm đánh giá hiệu năng.
8	OMPC	Sử dụng trình biên dịch ứng dụng thư viện MPI tại thời điểm chạy	Cài đặt đầy đủ chỉ thị của OpenMP	Đang trong quá trình nghiên cứu bước đầu thực nghiệm hệ thống tăng tốc chưa ổn định.
9	CAPE	Sử dụng kỹ thuật chụp ảnh tiến trình để tự động hoá mô hình thực hiện và mô hình bộ nhớ của OpenMP	Hiệu năng cao	Chưa cài đặt hoàn chỉnh tất cả các chỉ thị của OpenMP Chưa khai thác tối đa khả năng đa luồng trên mỗi nút phụ

1.7 CAPE đơn luồng

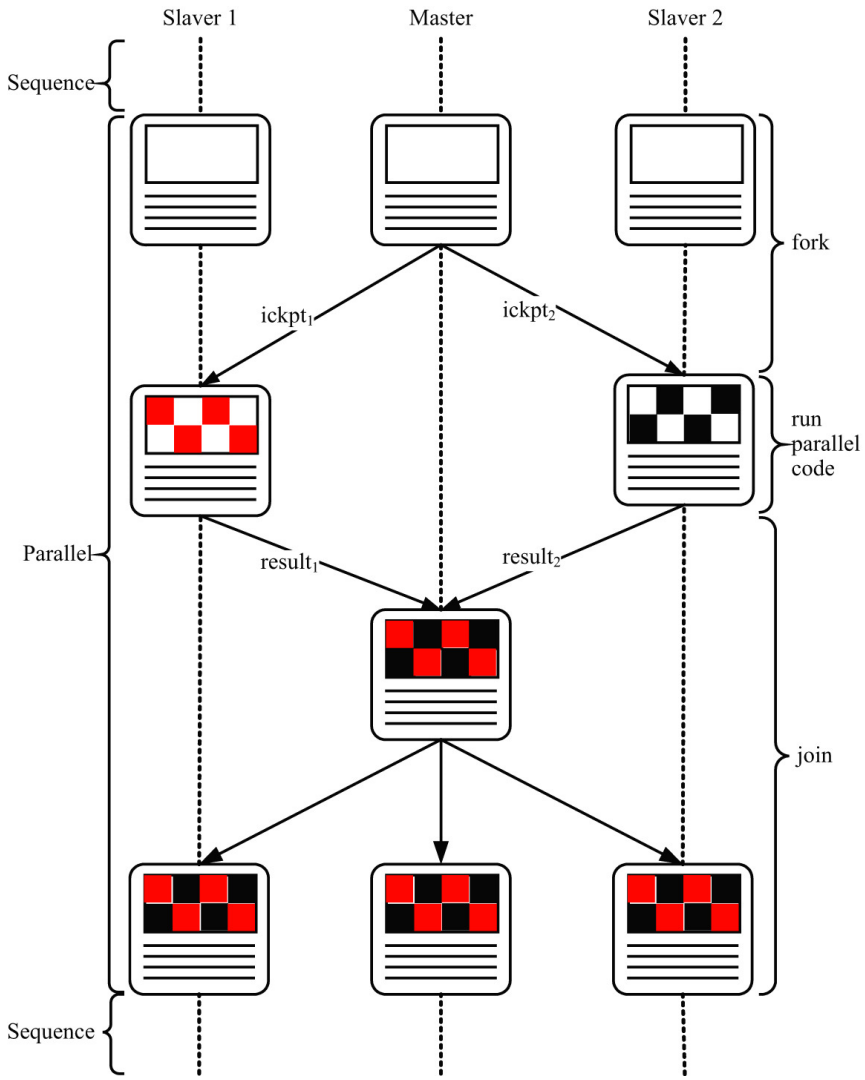
1.7.1 Giới thiệu chung về CAPE đơn luồng

CAPE (Checkpointing-Aide Parallel Executing) là một phương pháp dựa trên kỹ thuật chụp ảnh tiến trình, có mục tiêu cài đặt API OpenMP trên các hệ thống sử dụng bộ nhớ phân tán [12]-[20]. CAPE được GS. Eric Renault phát minh về mặt nguyên lý và sau đó được TS. Hà Viết Hải, TS. Trần Văn Long phát triển, thử nghiệm trong thời gian làm luận án tại Viện Viễn thông Nam Paris (Pháp). Tất cả các công việc cơ bản để thực hiện các cấu trúc song song OpenMP, bao gồm tổ chức nhóm làm việc, phân chia công việc, trích rút kết quả,… đều được thực hiện với công cụ cơ sở là kỹ thuật chụp ảnh tiến trình. CAPE hoạt động theo mô hình Chính – Phụ (Master – Slave): Chương trình OpenMP (đã được biên dịch sang dạng của CAPE) được thực hiện trên một tập các máy tính nối mạng, trong đó có 1 nút chính (master) chứa các kết quả thực hiện chương trình và các nút nút_phụ/ nút_tính_toán (slave) thực hiện song song các phần khác nhau của các câu lệnh song song OpenMP. Trong những phiên bản CAPE được tiến hành ở các nghiên cứu trước luận án này, các phần việc thực hiện tại các nút phụ được đảm trách bởi một tiến trình đơn luồng. Vì vậy, trong luận án này chúng tôi gọi chung CAPE ở các phiên bản trước là CAPE đơn luồng.

1.7.2 Mô hình hoạt động của CAPE đơn luồng

Để cài đặt được API OpenMP một cách tương tích hoàn toàn trên hệ thống bộ nhớ phân tán, cần phải cài đặt được toàn bộ các yêu cầu về mô hình hoạt động, mô hình bộ nhớ, các chỉ thị và những yêu cầu khác của OpenMP.

Về mô hình hoạt động, CAPE đưa ra mô hình mới trong đó đảm bảo mô hình *fork-join* của OpenMP về mặt lôgic nhưng với một số thay đổi để thực hiện được trên các hệ thống máy tính sử dụng bộ nhớ phân tán. Mô hình này được minh hoạ trong Hình 1.6.



Hình 1.6. Mô hình hoạt động của CAPE đơn luồng

Điểm khác nhau quan trọng giữa mô hình thực hiện CAPE và OpenMP, CAPE sử dụng tiến trình (process) thay cho luồng (thread) của OpenMP. Sự thay thế này giúp cho các câu lệnh song song của OpenMP có thể được tổ chức trên hệ thống bộ nhớ phân tán, bởi vì không thể tổ chức mô hình một tiến trình đa luồng chạy trên các nút khác nhau (máy tính khác nhau) của hệ thống này. Hình 1.6 minh họa mô hình thực hiện của CAPE trong tình huống chạy trên 3 nút, trong đó có một nút chính và hai nút phụ. Nút chính đóng vai trò như luồng chính ban đầu trong chương trình OpenMP và hai nút phụ đóng vai trò là các luồng tính toán khi gặp các chỉ thị xử lý song song trong OpenMP. Đầu tiên, chương trình ứng dụng được khởi tạo và chạy trên tất cả các nút. Khi gặp đoạn mã song song, nút chính thực hiện chia công việc và phân phối công việc đến các nút phụ theo dạng một ảnh chụp tiến trình gia tăng rời rạc (DICKPT) [12][13]. Mỗi nút phụ nhận được một ảnh chụp tiến trình, tích hợp vào vùng nhớ của nó và chạy thực hiện công việc

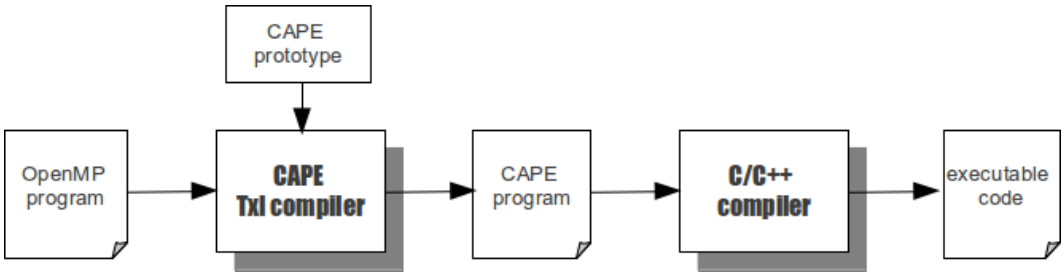
được phân công. Sau đó, kết quả tính toán trên mỗi nút phụ được trích rút gửi về nút chính cũng thông qua một ảnh chụp tiến trình. Nút chính nhận được tất cả các ảnh chụp tiến trình từ các nút phụ, thực hiện tổng hợp và tích hợp vào không gian nhớ của nó rồi gửi lại ảnh chụp tiến trình mới đến tất cả nút phụ. Các nút phụ nhận ảnh chụp tiến trình mới tích hợp vào không gian nhớ của chúng, sau bước này, cả nút phụ và nút chính đều có trạng thái không gian nhớ giống nhau, vì thế, chúng xem như đều đã thực hiện xong phần việc của đoạn mã song song này và sau đó tiếp tục chạy các đoạn mã tiếp theo của chương trình.

1.7.3 Biên dịch mã từ OpenMP sang CAPE

Để thực hiện chương trình OpenMP, trước hết cần được *khử-OpenMP* – tức là biên dịch sang dạng chương trình không còn các chỉ thị OpenMP nữa. Nếu chương trình đích sẽ chạy trên các hệ thống sử dụng bộ nhớ chia sẻ, công việc này được thực hiện bởi các chương trình biên dịch OpenMP đã có sẵn.

Đối với CAPE, một chương trình biên dịch chuyên biệt đã được phát triển bằng ngôn ngữ *txl* [70][71], dạng dịch mã-nguồn sang mã-nguồn (C sang C) để chuyển các chỉ thị OpenMP sang dạng các câu lệnh C thuần tuý, trong đó có sử dụng một thư viện các hàm CAPE. Những hàm này thực hiện nhiệm vụ của DICKPT bao gồm những chức năng cơ bản như: theo dõi tiến trình được chụp ảnh, chụp ảnh tiến trình, phân phối ảnh chụp tiến trình, gửi nhận ảnh chụp tiến trình, trích rút/tích hợp ảnh chụp tiến trình vào vùng không gian nhớ tiến trình,… Ngoài ra, còn có một tập hợp các khuôn dạng đã được xác định trong trình biên dịch CAPE để tự động thực hiện dịch chương trình OpenMP sang chương trình CAPE để chạy được trên nền tảng CAPE. Sau khi đã được biên dịch bởi chương trình dịch của CAPE, mã nguồn tiếp tục được biên dịch sang dạng mã máy bởi một trình biên dịch C thông thường.

Quy trình biên dịch được mô tả trong Hình 1.7 [17].



Hình 1.7. Quy trình biên dịch chương trình OpenMP thành chương trình CAPE

Trong quy trình trên, các hàm CAPE (dạng mã C) được bổ sung tự động vào chương trình dạng CAPE. Dưới đây là những hàm cơ bản trong thư viện này.

- `start( )` bắt đầu giám sát trình ứng dụng
- `stop( )` dừng việc giám sát trình ứng dụng
- `create(ckpt)` tạo ảnh chụp tiến trình và lưu ra `ckpt`
- `inject(ckpt)` tích hợp ảnh chụp tiến trình vào vùng nhớ của ứng dụng
- `send(ckpt, node)` gửi một ảnh chụp tiến trình `ckpt` đến `node`
- `wait_for(ckpt)` đợi nhận một ảnh chụp tiến trình
- `merge(ckpt1, ckpt2)` trộn (merge) ảnh chụp tiến trình `ckpt2` vào ảnh chụp tiến trình `ckpt1`
- `broadcast(ckpt)` gửi `ckpt` đến tất cả các nút slave
- `receive(ckpt)` nhận ảnh chụp tiến trình `ckpt`

Đối với 1 chỉ thị OpenMP, cần có một khuôn dạng chuyển đổi tương ứng. Hình 1.8 [17] trình bày khuôn dạng dành cho cấu trúc `omp parallel for` – cấu trúc song song cơ bản nhất của OpenMP. Để đơn giản cho việc trình bày, giả sử số bước lặp của cấu trúc `for` bằng với số nút slave.

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

automatically translated into

```
1 if ( master () ){
2       start ()
3       for ( A ; B ; C ){
4               create ( before )
5               send ( before, slaver_i )
6       }
7       create ( final )
8       stop ()
9       wait for ( after )
10      inject ( after )
11      if ( ! last parallel () ){
12              merge (final , after )
13              broadcast ( final )
14      }
15 }
16 else {
17      receive ( before )
18      inject ( before )
19      start ()
20      D ;
21      create ( after_i )
22      stop ()
23      send (after_i, master )
24      if ( ! last parallel () ){
25              receive ( final )
26              inject ( final )
27      }
28      else
29              exit
30 }
```

Hình 1.8. Khuôn dạng biên dịch cấu trúc **`omp parallel for`** trong CAPE đơn luồng

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

Ở tiến trình chính (câu lệnh 1-15): phần đầu của vòng lặp `for` được thực hiện ở trên tiến trình này. Tại mỗi bước lặp, một ảnh chụp tiến trình 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 `for` được 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 16-30): 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 vào không gian nhớ của nó. Như vậy, sau bước này, tiến trình `i` đã 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 20-21). Sau khi thực hiện xong việc tính toán, một ảnh chụp tiến trình được tạo ra để trích rút các kết quả thực hiện được bởi thân của vòng lặp. Ảnh chụp này được gửi về tiến trình chính. Nếu đây chưa phải là cấu trúc song song cuối cùng của chương trình, tiến trình phụ nhận ảnh chụp 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.

Đối với các cấu trúc song song khác với cấu trúc `parallel for` nói trên, có hai hướng để thực hiện việc biên dịch: 1) xây dựng các khuôn dạng chuyển đổi chuyên biệt cho cấu trúc đó; và 2) trước hết chuyển đổi cấu trúc này sang dạng cấu trúc `parallel for`, sau đó tiếp tục dùng khuôn dạng của `parallel for` để tiếp tục biên dịch. Hình 1.9 và Hình 1.10 lần lượt trình bày ví dụ cho việc chuyển đổi cấu trúc `omp parallel section` một cách trực tiếp sang dạng CAPE và sang dạng `parallel for`.

```
# pragma omp parallel sections
{
    # pragma omp section
    P0 ;
    # pragma omp section
    P1 ;
    # pragma omp section
    P2 ;
}
```

↓ automatically translated into ↓

```
1   if ( master ( ) ) {
2        wait_for ( after ) ;
3        inject ( after ) ;
4        if ( !last_parallel ( ) )
5             broadcast ( after ) ;
6   } else {
7        switch ( get_process_num ( ) ) {
8             case 1 :
9                  start ( ) ;
10                 P0 ;
11                 create ( after0 ) ;
12                 stop ( ) ;
13                 send ( after0, master ) ;
14                 break ;
15            case 2 :
16                 start ( ) ;
17                 P1 ;
18                 create ( after1 ) ;
19                 stop ( ) ;
20                 send ( after1, master )
21                 break ;
22            case 3 :
23                 start ( ) ;
24                 P2 ;
25                 create ( after2 ) ;
26                 stop ( ) ;
27                 send ( after2, master ) ;
28                 break ;
29       }
30       if ( !last_parallel ( ) ) {
31            receive ( after ) ;
32            inject ( after ) ;
33       } else
34            exit ( ) ;
35  }
```

Hình 1.9. Khuôn dạng chuyển đổi cho cấu trúc **`omp parallel section`** sang dạng CAPE

```
# pragma omp parallel sections
{
#   pragma omp section
    P0 ;
#   pragma omp section
    P1 ;
#   pragma omp section
    P2 ;
}
```

↓ can be translated into ↓

```
# pragma omp parallel for
    for ( i = 0 ; i < 3 ; i ++ ) {
        switch ( i ) {
        case 0 :
            P0 ;
            break ;
        case 1 :
            P1 ;
            break ;
        case 2 :
            P2 ;
        }
    }
```

Hình 1.10. Khuôn dạng chuyển đổi cấu trúc **`omp parallel section`** sang cấu trúc **`omp parallel for`**

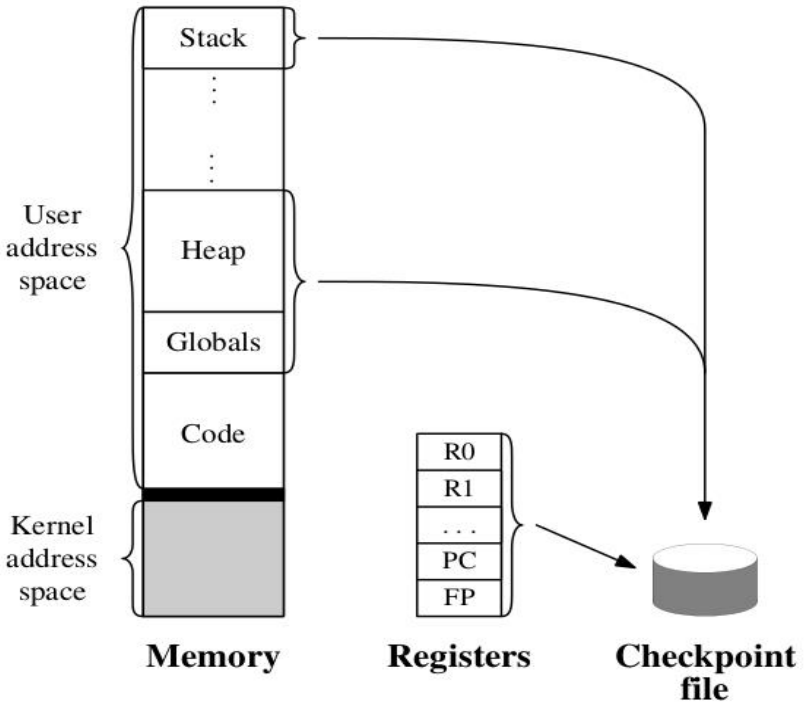
Hiện tại, CAPE chưa hỗ trợ cho các dạng chỉ thị OpenMP lồng nhau. Tuy nhiên, xét về nguyên lý không có hạn chế nào để ngăn trở vấn đề này và nó có thể tiếp tục được nghiên cứu để đáp ứng một cách đầy đủ.

1.7.4 Kỹ thuật chụp ảnh tiến trình áp dụng cho CAPE đơn luồng

Chụp ảnh tiến trình [35] là kỹ thuật chụp và lưu trữ trạng thái của một tiến trình đang vận hành sao cho có khả năng khôi phục lại trạng thái của tiến trình ở các thời điểm sau đó. Khi một chương trình được chạy trạng thái của chúng được thể hiện qua các giá trị trong không gian nhớ (memory space) của tiến trình, giá trị trong các thanh ghi và trạng thái của hệ điều hành. Không gian nhớ của tiến trình thường được phân chia thành 04 phân vùng: Code, Globals, Heap và Stack (ngoài phân vùng dành cho hệ điều hành). Hình 1.11 minh họa các thông tin cần được chụp và lưu trữ trạng thái của một tiến trình trên bộ vi xử lý đơn lõi. Phần vùng nhớ Kernel được sử dụng bởi hệ điều hành và vùng

Code chứa chương trình thường không thay đổi nên ảnh chụp tiến trình chỉ lưu trạng thái của vùng Stack, Heap và Globals.

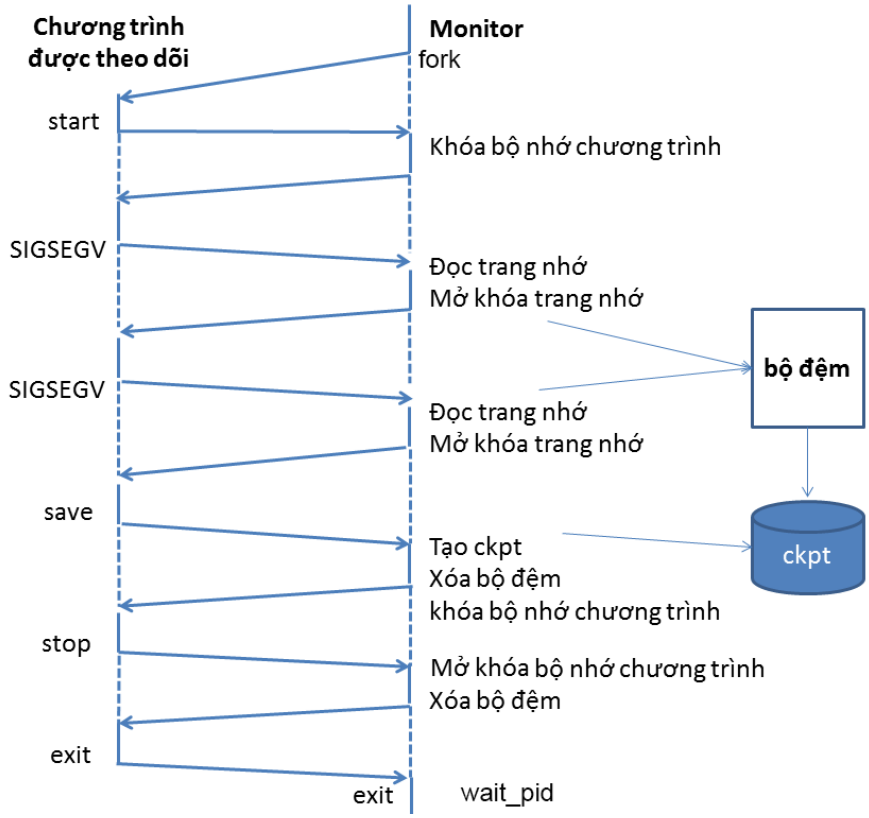
Có thể phân chia kỹ thuật chụp ảnh tiến trình thành hai loại chính. Chụp ảnh toàn bộ (Complete Checkpointing) trong đó toàn bộ bộ nhớ được lưu trữ vào mỗi ảnh chụp và Chụp ảnh gia tăng (Incremental Checkpointing) trong đó chỉ những phần bộ nhớ có thay đổi giá trị so với lần chụp ảnh trước đó được lưu trữ vào ảnh chụp mà thôi.



Hình 1.11. Minh họa các thông tin được lưu trữ trong 1 ảnh chụp tiến trình [35]

Để phục vụ một cách tối ưu cho CAPE, Kỹ thuật chụp ảnh tiến trình gia tăng rời rạc DICKPT đã được TS. Hà Viết Hải phát triển vào những năm 2009-2011 [13]. Kỹ thuật này dựa trên cơ sở kỹ thuật chụp ảnh tiến trình gia tăng và cung cấp thêm khả năng chụp ảnh từng phần riêng biệt tương ứng với từng đoạn mã rời rạc của chương trình.

Thời điểm chụp ảnh và gửi nhận ảnh chụp tiến trình là thời điểm chương trình viết theo CAPE gọi các hàm `create(before)`, `send (before,slaveri)`, `receive(before)`, `create(afteri)`, `send(after,master)`, `wait for(after)` như đã trình bày trong Hình 1.8.



Hình 1.12. Nguyên tắc hoạt động của của Kỹ thuật chụp ảnh tiến trình gia tăng rời rạc

Hình 1.12 mô tả cơ chế thực hiện chụp ảnh tiến trình của DICKPT, mỗi tiến trình có thể được chụp ảnh bởi một monitor. Monitor chịu trách nhiệm về khởi tạo tiến trình ứng dụng (tiến trình cần được theo dõi và chụp ảnh), nhận tín hiệu từ tiến trình chụp ảnh, tạo ra file ảnh tiến trình hoặc/và khôi phục lại trạng thái tiến trình từ các file ảnh tiến trình và chờ tín hiệu kết thúc của tiến trình.

Sau khi bắt đầu thực thi chương trình được theo dõi, monitor đợi tín hiệu để chụp ảnh tiến trình. Có 5 trường hợp chính: 3 trong số đó liên quan đến 3 chỉ thị chụp ảnh tiến trình và 2 trường hợp khác liên quan đến tín hiệu ngắt (interrupt) `SIGSEGV` và tín hiệu `exit`.

a. Tín hiệu `start`: Tiến trình ứng dụng được tạm ngừng hoạt động để thiết lập quyền truy cập chỉ-đọc cho tất cả các trang nhớ của chính nó. Sau khi thực hiện xong việc này, tiến trình ứng dụng tiếp tục thực hiện các đoạn mã tiếp theo của mình, trong trạng thái sẵn sàng để được chụp ảnh. Từ lúc đó, bất kỳ tín hiệu `SIGSEGV` được tạo ra bởi tiến trình ứng dụng đang được theo dõi sẽ được gửi đến và xử lý bởi monitor. Có hai trường hợp xảy ra:

Tín hiệu `SIGSEGV` được sinh ra bởi vì chương trình muốn ghi lên một trang nhớ có trạng thái chỉ-đọc. Điều này có nghĩa rằng trang nhớ này hợp lệ và lần đầu tiên được truy cập kể từ lần tín hiệu `start` trước đó được tạo ra. Kết quả, nội dung của trang được đọc bởi monitor bằng cách sử dụng hàm `read_range()` của DICKPT driver và được lưu trữ vào một vùng nhớ đệm trong monitor để làm căn cứ tạo ảnh chụp tiến trình sau này, sau đó thuộc tính cho phép ghi được thiết lập trang nhớ tương ứng và tiến trình được theo dõi được yêu cầu tiếp tục thực hiện chương trình.

Trong các trường hợp còn lại, tín hiệu `SIGSEV` được tạo ra do trang nhớ không tồn tại trong không gian địa chỉ ảo của chương trình được theo dõi hoặc quyền truy cập ban đầu của trang nhớ không có khả năng ghi. Khi đó, tín hiệu `SIGSEGV` không được tạo ra bởi vì monitor thay đổi quyền truy cập của các trang nhớ này mà bởi một lý do khác bao gồm lỗi bên trong chương trình. Đối với trường hợp này, monitor không xử lý tín hiệu `SIGSEGV`. Nếu có cơ chế được thiết lập bên trong chương trình để bắt tín hiệu này, việc xử lý liên quan được thực thi. Nếu không, nó được trả về cho hệ điều hành.

b. Sau tín hiệu `SIGSEGV`, có hai trường hợp liên quan đến kỹ thuật DICKPT:

- **Tín hiệu `save`**: Khi monitor nhận được một yêu cầu để tạo ra một ảnh chụp tiến trình của tiến trình được theo dõi. Khi đó, monitor sẽ thực hiện các bước như sau:

1) Đọc nội dung hiện thời của tất cả các trang đã được sửa đổi kể từ ảnh tiến trình trước đó hoặc từ lúc bắt đầu chương trình, sử dụng hàm `read_range()` của DICKPT driver.
2) Với mỗi trang nhớ đã đọc ở trên, so sánh các giá trị trong các ô nhớ hiện tại với các giá trị tương ứng trong trang nhớ đã được lưu trữ trước đó để tìm ra chính xác những ô nhớ đã thay đổi giá trị và lưu trữ các giá trị đã được thay đổi này vào trong ảnh chụp tiến trình đang được tạo.
3) Thiết lập tất cả các trang nhớ về trạng thái chỉ-đọc để chuẩn bị cho lần chụp ảnh gia tăng tiếp theo. Đồng thời các trang nhớ đã được lưu trữ trong bộ nhớ đệm của monitor cũng bị xóa.

- **Tín hiệu `stop`:** Tiến trình được theo dõi được tạm ngưng để thiết lập quyền truy cập tới trạng thái cho phép ghi trên tất cả các trang nhớ của tiến trình.

Sau đó monitor không chịu trách xử lý tín hiệu `SIGSEGV` từ chương trình được theo dõi.

c. Tín hiệu `exit`: Khi tiến trình được theo dõi kết thúc việc thực thi của nó, có nghĩa là sau khi hàm `exit` của hệ thống được gọi, cho dù nó được gọi một cách tường minh hoặc không tường minh bằng mã lỗi là tiến trình được theo dõi không tồn tại được trả về, monitor hiểu rằng tiến trình bị theo dõi đã kết thúc vì vậy monitor cũng kết thúc việc chạy tiến trình của nó.

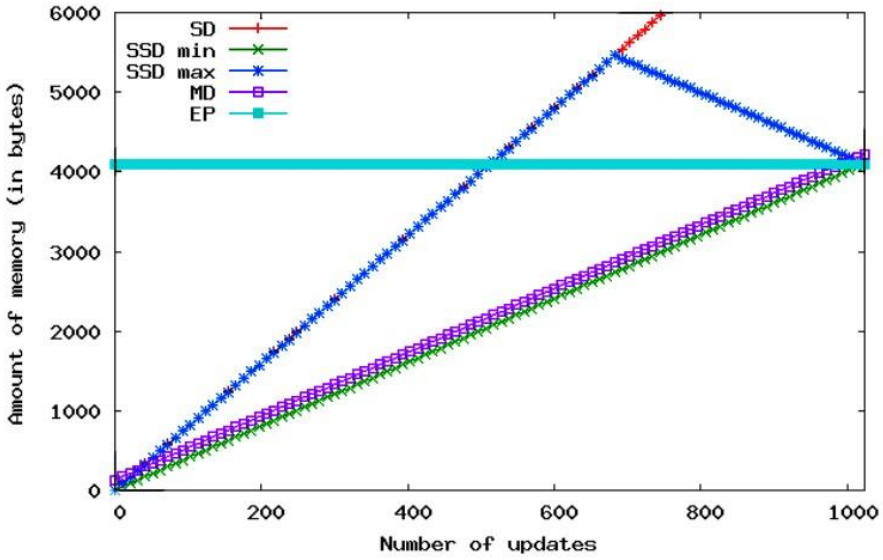
Trong một chương trình ứng dụng, có thể có nhiều cặp `start-stop` và trong mỗi cặp `start-stop`, có thể có nhiều chỉ thị `save`. Tính chất này cho phép monitor theo dõi và chụp ảnh từng phần rời rạc trong quá trình thực hiện của một tiến trình. Đây chính là đặc điểm khác biệt lớn nhất của Kỹ thuật chụp ảnh tiến trình gia tăng rời rạc so với Kỹ thuật chụp ảnh tiến trình gia tăng thông thường (Incremetal Checkpointing – ICKPT) được để cập ở các tài liệu [35][37][40][51][66][67].

1.7.5 Cấu trúc dữ liệu cho ảnh chụp tiến trình

Ảnh chụp tiến trình do DICKPT tạo ra chứa hai bộ dữ liệu: a) giá trị thanh ghi - tức là tất cả các giá trị thanh ghi tại thời điểm chụp, và b) không gian địa chỉ của chương trình được giám sát- tức là tất cả dữ liệu của chương trình được giám sát có sửa đổi so với chụp ảnh trước đó.

Phần lớn dữ liệu của ảnh chụp tiến trình là bộ dữ liệu lưu trữ dữ liệu thay đổi của chương trình.

Phần tổ chức và xử lý dữ liệu của ảnh chụp tiến trình là word (từ) (4- byte kích thước bộ nhớ để lưu trữ 1 từ). Trong [12] đã đề xuất 4 cách tổ chức bộ nhớ tối ưu kích thước của ảnh chụp tiến trình



Hình 1.13. Kích thước bộ nhớ cần thiết để lưu trữ ảnh chụp tiến trình [12]

- *Cấu trúc đơn từ - Single data (SD).* Với cấu trúc này mỗi phần tử được lưu theo định dạng 8 byte: 4 byte để lưu địa chỉ và 4 byte lưu dữ liệu được cập nhật tại địa chỉ đó. Cấu trúc của từng phần tử dữ liệu là `{(addr, value)}`.

- *Cấu trúc dữ liệu liên tiếp - Several successive data (SSD).* Dữ liệu được lưu theo cấu trúc {(addr, size, values)}. Trình chụp ảnh tiến trình sẽ đọc lưu thông tin dữ liệu từ địa chỉ addr đi size bytes để lưu vào ảnh chụp. Trong DICKPT, kích thước tối đa của `size` là 1 trang nhớ.

- *Cấu trúc sơ đồ ma trận - Many data (MD).* Dữ liệu được tổ chức theo cấu trúc `{(addr, map, values)}`. Ở đây, `map` là mảng kiểu bit để đánh dấu cho từng vùng nhớ (word). Tương ứng mỗi trang nhớ 4kB và kích thước map là 1024bit.

- *Cấu trúc toàn trang nhớ- Entire page (EP).* Trong cách này toàn bộ trang nhớ được lưu trữ. Do đó cấu trúc sẽ là `{(addr, values)}`. Ở đây `size` là kích thước một trang nhớ.

Hình 1.13 trình bày so sánh kích thước không gian bộ nhớ để lưu trữ ảnh chụp tiến trình theo 4 cấu trúc khác nhau [12]. SD, MD, và EP chỉ phụ thuộc vào số lượng vùng nhớ có cập nhật dữ liệu. SSD vừa phụ thuộc vào số lượng cập nhật và sự phân tán của các vùng nhớ được cập nhật dữ liệu. Trường hợp tốt nhất của SSD_{min} đạt được khi vùng dữ liệu được cập nhật là địa chỉ liên tục và trường hợp xấu nhất SSD_{max} xảy ra khi số các vùng dữ liệu được cập nhật là phân tán tối đa.

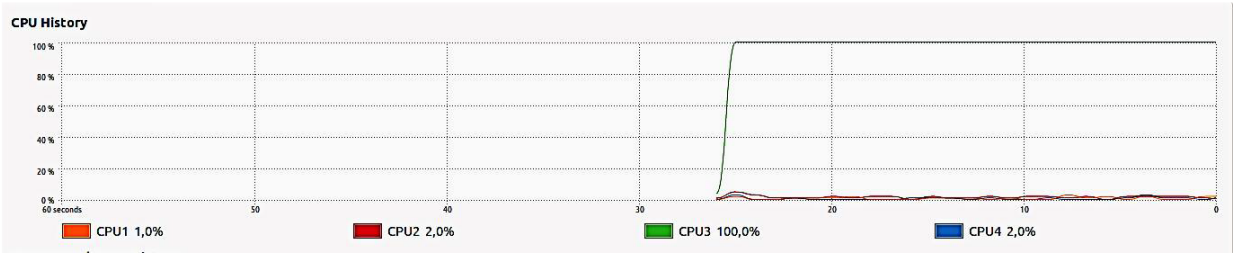
Mỗi cấu trúc tổ chức ảnh chụp tiến trình có ưu nhược điểm riêng tủy theo các tình huống thực tế của chương trình. Ví dụ, cấu trúc EP rất tốt cho trường hợp vùng nhớ của

tiến trình được thay đổi toàn trang nhớ; SD luôn tốt cho trường hợp chỉ vài vùng nhớ được cập nhật; MD tốt trong trường hợp có nhiều vùng nhớ được cập nhật dữ liệu và địa chỉ phân tán; trong khi đó SSD phù hợp cho tình huống số lượng cập nhật dữ liệu nhỏ hơn. Hiện tại, cấu trúc của DICKPT đang sử dụng là SSD.

Việc giải quyết tương tranh cũng như đồng bộ thời gian khi các nút phụ gửi các ảnh chụp tiến trình theo cấu trúc SSD về nút chính để cập nhật dữ liệu và đồng bộ dữ liệu theo các chỉ dẫn của OpenMP được trình bày chi tiết ở Chương 4.

1.7.6 Hạn chế của CAPE khi sử dụng trên hệ thống máy tính có CPU đa lõi

Ngày nay, đa lõi là kiến trúc CPU phổ biến, với số lượng lõi được tăng lên thường xuyên. Ngay cả máy tính phổ thông cũng được trang bị CPU có 4 đến 8 lõi và kết quả kéo theo các mạng máy tính sử dụng CPU đa lõi trở nên phổ biến. Tuy nhiên, trong mô hình thực hiện của CAPE thuộc các nghiên cứu trước đây, đặc điểm này chưa chú trọng khai thác. Vấn đề này thể hiện ở trong mô hình hoạt động, trong đó các cấu trúc song song của OpenMP được chuyển đổi để chạy trên nhiều nút slave nhưng mỗi nút chỉ chạy một tiến trình đơn luồng – tức là thực hiện ở dạng chương trình tuần tự. Điều này dẫn đến sự lãng phí tài nguyên của máy tính như trong Hình 1.14- chỉ rõ nút slave có CPU 4 lõi nhưng CAPE chỉ mới sử dụng được một lõi, các lõi khác đang ở trạng thái nghỉ.



Hình 1.14. Tỷ lệ sử dụng các lõi của CAPE đơn luồng trên các nút phụ

Rõ ràng, việc các lõi ở trạng thái nghỉ cần phải được khai thác để đạt được hiệu suất thực hiện chương trình cao hơn. Một cách trực quan, điều này có thể đạt được bằng cách cho thực hiện song song các đoạn mã tính toán tại các nút slave.

1.8 Tiểu kết Chương 1

Chương đầu tiên của luận án đã giới thiệu tổng quan về các phương pháp chuyển đổi API OpenMP trên hệ thống bộ nhớ phân tán. CAPE là một trong những phương pháp khả thi với những ưu điểm tiềm năng. Tuy nhiên, phiên bản CAPE hiện tại chỉ đang khai thác các mạng máy tính theo quan điểm sử dụng các bộ xử lý đơn lõi. Trong

bối cảnh hiện nay, khi hầu như tất cả các CPU của máy tính thường được sản xuất đều có cấu trúc đa lõi thì mô hình hoạt động của CAPE chưa khai thác hết hiệu suất của hệ thống.

Đó chính là cơ sở để luận án xác định được các mục tiêu cần nghiên cứu, cải tiến mô hình CAPE hiện tại để có thể nâng cao hiệu năng bằng cách khai thác hết công suất của các bộ vi xử lý đa lõi.

Các kết quả của chương này được công bố ở công trình [CT1].

CHƯƠNG 2 CAPE TRÊN HỆ THỐNG TÍNH TOÁN ĐA LÕI

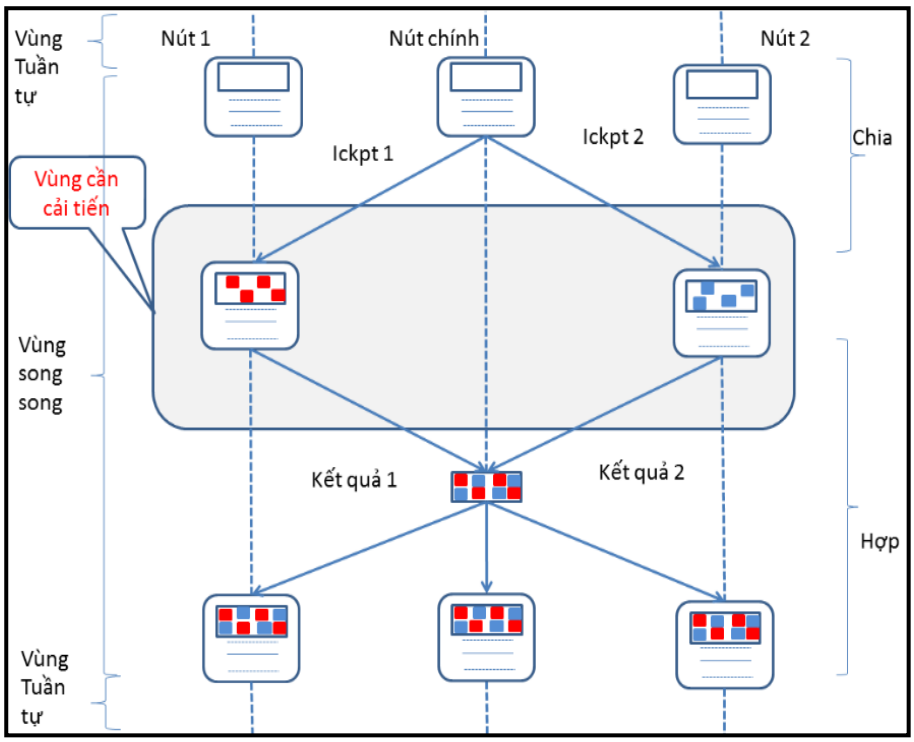
Trong chương I đã trình bày tổng quan các nghiên cứu liên quan, các kiến thức cơ bản liên quan đến vấn đề nghiên cứu như khái niệm tính toán hiệu năng cao, các hình thức, mức độ song song, hệ số tăng tốc, phân loại kiến trúc song song, CPU đa lõi, hỗ trợ của hệ điều hành đối với CPU đa lõi, đặc biệt giới thiệu chi tiết về đối tượng nghiên cứu chính của luận án CAPE đơn luồng để cải tiến.

Chương này sẽ trình bày chi tiết mô hình CAPE mới, song song hai mức giúp khai thác tốt tài nguyên của máy tính sử dụng CPU đa lõi và làm tăng hiệu năng của CAPE lên nhiều lần khi so sánh với phiên bản CAPE đơn luồng.

2.1 Nguyên lý chung

Để thực hiện ý tưởng khai thác tốt hơn các tài nguyên tính toán của các hệ thống máy tính có CPU nhiều lõi bằng cách tăng số lượng tiến-trình/luồng chạy song song trong mỗi nút phụ, cần có mô hình CAPE mới để thực hiện. Sau đây là chi tiết về phương pháp thực hiện cụ thể.

Theo mô hình hoạt động hiện tại của CAPE đơn luồng, như ở Hình 2.1, tại các vùng song song, mỗi nút phụ được giao một phần công việc của đoạn mã song song và thực hiện trên một tiến trình duy nhất. Chính điều này dẫn đến việc hầu như chỉ có một lõi CPU là được sử dụng với tỷ lệ cao, còn các lõi khác ở trạng thái nghỉ.



Hình 2.1. Mô hình hoạt động của CAPE và vùng cần cải tiến để khai thác khả năng của các bộ xử lý đa lõi

Để tăng được hiệu suất của chương trình, rõ ràng trong khoảng thời gian thực hiện tính toán ở các nút phụ, cần song song hóa các đoạn mã này, tức là phải tìm cách để chương trình được chạy bởi đa tiến trình hoặc đa luồng (song song mức thứ 2). Trên Hình 2.1, vùng khoanh tô màu nền đậm hơn chính là vùng cần cải tiến. Dựa theo nguyên tắc hoạt động của CAPE hiện tại, có 3 phương pháp có thể thực hiện điều này: 1) Sử dụng đa tiến trình trên mỗi nút phụ bằng cách cho thực hiện song song nhiều lần chương trình ứng dụng trên mỗi nút (Phương pháp 1); 2) Sử dụng song song trên mỗi nút phụ bằng cách cho thực hiện song song chương trình ứng dụng trên các máy ảo, tạo nhiều máy ảo trên từng máy thật (Phương pháp 2); và 3) Song song hóa đoạn mã tính toán trên nút phụ theo mô hình đa luồng, nghĩa là song song 2 mức: mức 1 liên quan đến phân chia và chạy đồng thời các tác vụ trên nhiều máy, trong khi mức 2 liên quan đến việc sử dụng đa luồng trên mỗi nút phụ để chạy song song phần công việc được chia cho nó (Phương pháp 3).

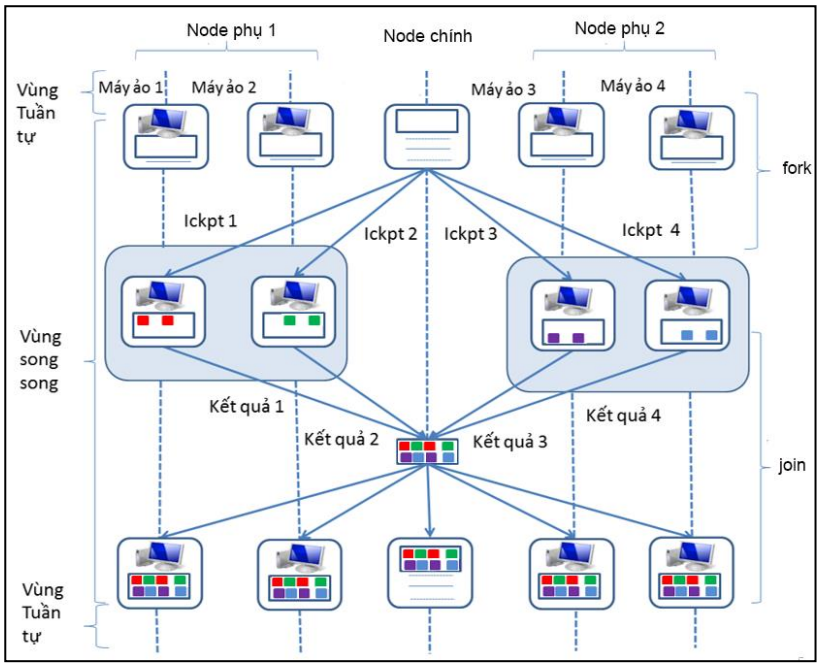
Phương pháp 1) đã được thử nghiệm và công bố kết quả ở [64]. Tuy nhiên với nhược điểm chính của phương pháp này là khả năng xảy ra các tương tranh giữa các chương trình trên cùng một nút vì chúng sử dụng đều có chung địa chỉ IP. Hơn nữa, việc sử dụng socket để cài đặt các cơ chế truyền dữ liệu cũng có thể gây nên các xung đột về

tài nguyên giữa các chương trình và gây lỗi với số tiến trình tính toán lớn, số lượng lỗi dẫn đến việc dừng hệ thống lên đến đến gần 30% trường hợp thử nghiệm. Đồng thời, việc sử dụng nhiều tiến trình độc lập của cùng một chương trình đòi hỏi các nút tính toán phải có bộ nhớ RAM lớn để đảm bảo được tốc độ tính toán. Do vậy, cần phải thực hiện các phương pháp khác tối ưu hơn. Luận án này đã thực hiện nghiên cứu và thực nghiệm chi tiết cả phương pháp 2) và 3) chi tiết được trình bày trong phần tiếp theo.

2.2 Mô hình CAPE song song hai mức dựa trên phương pháp sử dụng nhiều máy ảo

2.2.1 Ý tưởng và cách thực hiện

Phương pháp này được đưa ra với ý tưởng chính là tránh sự tranh chấp tài nguyên dẫn đến dừng hệ thống của Phương pháp 1 như đã trình bày trong mục 2.1 ở trên. Trong đó, trình ứng dụng của người dùng và trình theo dõi được cài đặt và thực hiện trên một máy ảo và có nhiều máy ảo như vậy được cài đặt và vận hành song song trên mỗi máy vật lý của các nút phụ. Sự khác biệt so với Phương pháp 1 là sự độc lập của các tiến trình trên các máy ảo, mỗi máy ảo có địa chỉ IP riêng, làm giảm nguy cơ xung đột giữa các tiến trình do tranh chấp địa chỉ IP giữa các socket. Hình 2.2 trình bày mô hình hoạt động của phương pháp này gồm hai nút phụ (máy thật) mỗi nút phụ chạy 2 máy ảo, khi đó cả 4 máy ảo này được xem như là 4 nút phụ logic của mô hình CAPE. Như vậy trong mô hình này mức song song thứ hai được thực hiện bằng cách tạo nhiều máy ảo trên từng máy thật (máy vật lý).



Hình 2.2. Mô hình sử dụng nhiều máy ảo trên mỗi nút phụ

Để cài đặt theo phương pháp này một cách hiệu quả, cần sử dụng công cụ máy ảo có khả năng chạy được nhiều tiến trình trên cùng một máy vật lý, mỗi tiến trình có thể được xem như là một máy độc lập. Cụ thể là phải đáp ứng các yêu cầu:

- Chạy được nhiều máy ảo trên cùng một máy vật lý;
- Có khả năng thiết lập IP độc lập cho mỗi máy ảo, từ đó có khả năng kết nối các tiến trình trên máy ảo với các máy vật lý khác với máy đang vận hành chúng (host);
- Cung cấp hiệu năng tốt.

Hiện tại, có nhiều máy ảo có thể đáp ứng các yêu cầu trên, trong đó nổi bật là Xen, Open VZ và KVM. Đây đều là các máy ảo mã nguồn mở, tương thích với Linux, do đó dễ dàng được dùng để thử nghiệm với CAPE.

Chúng tôi đã cài đặt theo phương pháp này với máy ảo KVM (Kernel-based Virtual Machine) là một máy ảo hóa đầy đủ cho Linux trên nền tảng phần cứng x86. KVM cung cấp một kiến trúc ảo hóa căn bản và một module xử lý chuyên biệt. Trình quản trị KVM cho phép thực hiện nhiều máy ảo trên cùng một máy tính vật lý, trong đó mỗi máy có một địa chỉ IP khác nhau và điều này là rất hữu hiệu cho việc cấu hình CAPE. Nhờ khả năng này mà ta có thể dễ dàng chạy nhiều tiến trình của người dùng trên mỗi nút phụ mà không bị các xung đột về IP. Thực tế việc cài đặt khá dễ dàng, chỉ cần cài đặt CAPE và trình ứng dụng trên một máy ảo, sau đó nhân bản thành nhiều bản trên cùng một máy.

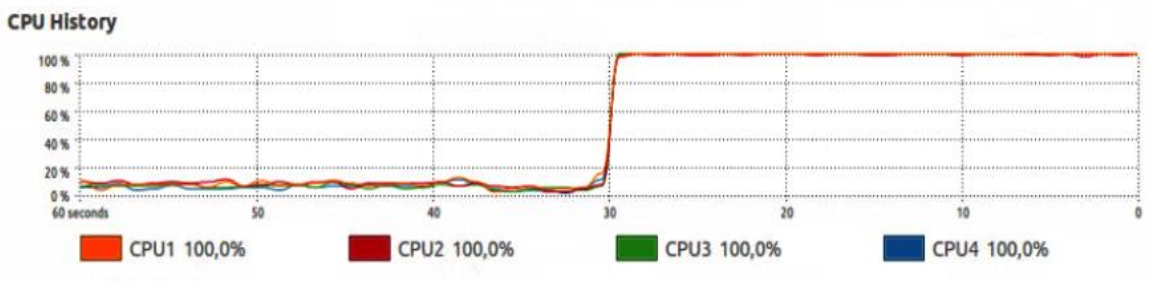
Công việc còn lại chỉ là viết lại Trình phân phối để cấu hình CAPE trên các máy ảo này. Cách thức cài đặt chi tiết được mô tả cụ thể trong Phụ lục 1.

2.2.2 Đánh giá hiệu năng

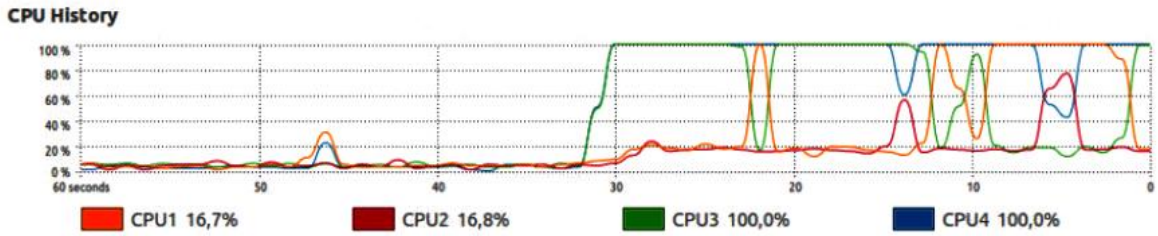
Chúng tôi thực hiện việc đánh giá mô hình CAPE theo phương pháp sử dụng máy ảo này trên một Cluster các máy tính để bàn có CPU 2 lõi x 2 siêu phân luồng (được tính như 4 lõi), nối mạng Ethernet 100 Mb/s.

2.2.2.1 Đánh giá tỷ lệ sử dụng CPU

Về khả năng khai thác các lõi trên hệ thống máy tính sử dụng CPU đa lõi, chúng tôi thu được kết quả tương tự như với trường hợp sử dụng đa tiến trình trên máy thực của Phương pháp 1. Trong đó, tỷ lệ khai thác 4 lõi của CPU là 100% khi sử dụng 4 máy ảo trên mỗi nút vật lý và 50% trong trường hợp sử dụng 2 máy ảo. Biểu đồ đo tỷ lệ sử dụng được chụp ảnh và trình bày lần lượt trong Hình 2.3 và Hình 2.4.



Hình 2.3. Tỷ lệ khai thác các lõi khi chạy 4 máy ảo

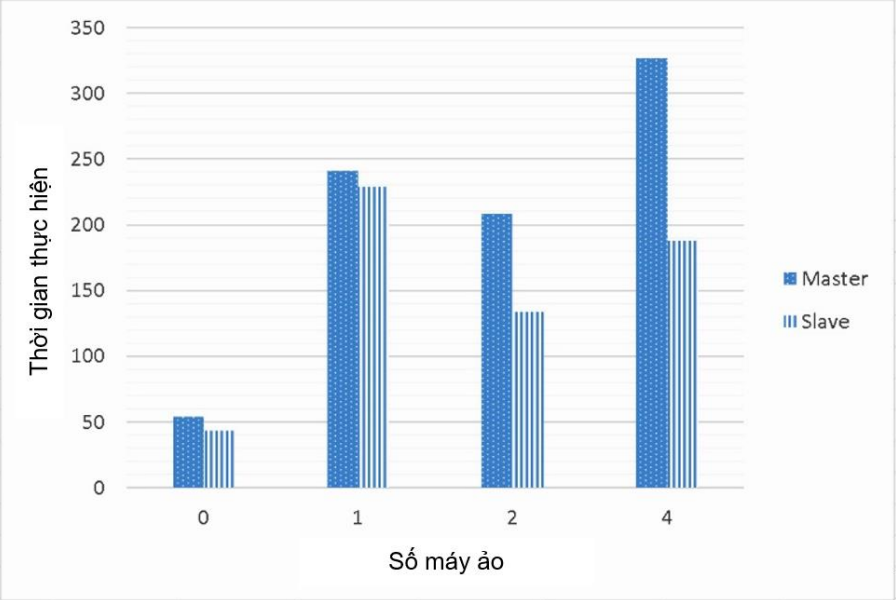


Hình 2.4. Tỷ lệ khai thác các lõi khi chạy 2 máy ảo

2.2.2.2 Đánh giá hiệu năng thực hiện chương trình

Để đánh giá phương pháp này về mặt hiệu năng, luận án đã thử nghiệm trên bài toán nhân 2 ma trận với kích thước ma trận là 6000x6000, sử dụng 21 nút vật lý (1 nút chính và 20 nút phụ (máy thật), số lượng máy ảo trên mỗi nút phụ biến thiên với các giá trị 1, 2 và 4 (do hệ thống máy tính thực nghiệm sử dụng CPU Intel Core i3 chỉ có 4 core logic). Thời gian thực hiện được so sánh với giải pháp CAPE hiện tại, tức là chạy tiến trình trên máy thực. Biểu đồ so sánh các thời gian chạy được trình bày trên Hình 2.5 trong đó nhóm 0 tương ứng với trường hợp chạy 1 tiến trình trên máy thực.

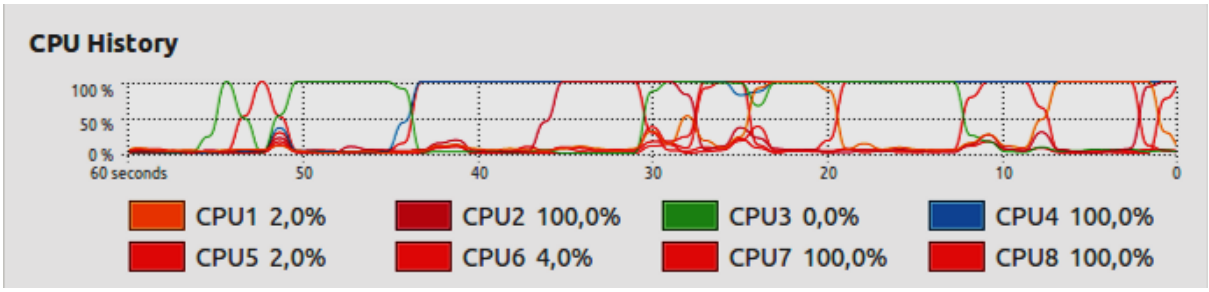
Như biểu đồ thể hiện, mặc dù CPU làm việc nhiều hơn, kết quả về mặt hiệu năng là không như mong đợi, thời gian thực hiện trong trường hợp sử dụng máy ảo luôn lớn hơn thời gian khi sử dụng máy thực (vật lý). Ngay cả trong trường hợp tốt nhất là sử dụng 2 máy ảo trên mỗi nút phụ, thời gian trên nút phụ và nút chính tăng lên khoảng 3 đến 4 lần. Điều này gây ra là do ảnh hưởng của chi phí sử dụng máy ảo lên thời gian thực hiện và trong trường hợp này, chi phí này là lớn hơn nhiều so với lợi ích thu lại khi chạy nhiều tiến trình của của trình ứng dụng trên các máy ảo này.



Hình 2.5. Thời gian thực hiện chương trình với số máy ảo khác nhau

Để kiểm chứng lại việc giảm hiệu năng của hệ thống khi sử dụng máy ảo chúng tôi đã thử nghiệm chạy một chương trình tuần tự nhân hai ma trận kích thước 6000x6000 viết bằng ngôn ngữ C chạy trên một máy tính cài hệ điều hành Ubuntu 14.04, máy ảo KVM, có cấu hình CPU Core i7, 4 lõi x 2 siêu phân luồng (được tính như 8 lõi), RAM 8 GB có cài các máy ảo với các kịch bản khác nhau để kiểm chứng: 1) Kịch bản chạy chương trình trên máy thật; 2) Kịch bản trên máy 1 máy ảo sử dụng cả 8 lõi CPU; 3) Kịch bản chạy trên 2 và 4 máy ảo mỗi máy ảo sử dụng 2 lõi CPU; 4) Kịch bản chạy trên 6 vào 8 máy ảo sử dụng mỗi máy ảo 01 lõi CPU. Trong tất cả các kịch bản sử dụng máy ảo, dung lượng RAM được phân đều cho các máy ảo. Để đơn giản trong việc thử nghiệm ở kịch bản sử dụng nhiều máy ảo với chương trình được lập trình tuần tự, chúng tôi chạy đồng thời chương trình nhân ma trận kích thước 6000x6000 ở các máy ảo. Ví dụ: Trường hợp chạy 2 máy ảo nếu thời gian chạy trên mỗi máy ảo tương đương với máy thật, có nghĩa đã tăng tốc được khoảng gấp đôi. Kết quả thực nghiệm với số liệu như ở Bảng 2.1

cho thấy mức độ suy giảm hiệu năng khi sử dụng máy ảo trong trường hợp này khá lớn, đối với trường hợp tốt nhất sử dụng 4 máy ảo mức độ suy giảm tới khoảng 1.8 lần, trong khi đó CPU vẫn thường xuyên sử dụng tối đa 100% công suất của 4 lõi và có sự luôn phiên sử dụng các lõi CPU theo thời gian như Hình 2.4.



Hình 2.6. Trạng thái sử dụng CPU khi chạy chương trình trên 4 máy ảo

Bảng 2.1. Kết quả thực nghiệm mức độ giảm hiệu năng khi sử dụng máy ảo

Số máy ảo	Kích thước ma trận	Tổng thời gian thực hiện chương trình	Tổng thời gian/số lượng máy	% suy giảm hiệu năng so với máy thật	Ghi chú
a	b	c	d	e	f
0	6000	954	954	0%	Máy thật
1	6000	4,542	4,542	476%	01 máy ảo
2	6000	5,172	2,586	271%	02 máy ảo
4	6000	6,829	1,707	179%	04 máy ảo
6	6000	10,715	1,786	187%	06 máy ảo
8	6000	21,755	2,719	285%	08 máy ảo

Lưu ý: trường hợp số máy ảo cột a =0 có nghĩa là chạy trên máy thật (vật lý). Đồng thời số core sẽ biến thiên trong mỗi trường hợp. Ví dụ như trường hợp số máy ảo là 4, mỗi máy ảo sẽ được cấu hình dùng 2 core. Tương tự hai máy ảo, mỗi máy ảo được cấu hình tài nguyên là 4 core CPU.

2.2.2.3 Đánh giá ưu điểm và nhược điểm của giải pháp

Tương tự như với giải pháp sử dụng đa tiến trình trên máy thực, ưu điểm nổi bật của giải pháp này là tính đơn giản, hầu như mã chương trình đều được giữ nguyên (trừ chỉnh sửa trong Trình phân phối). Tất nhiên là giải pháp này có phức tạp hơn do phải có thêm một bước là cài đặt và sử dụng các máy ảo, tuy nhiên việc này không có vấn đề gì về mặt kỹ thuật, chỉ cần sử dụng một máy ảo sẵn có. Ưu điểm thứ 2 là ở mỗi tiến trình được cài đặt trên mỗi máy ảo có IP khác nhau nên tránh được xung đột và các lỗi gây ra do xung đột, một lỗi thường xảy ra trong giải pháp sử dụng nhiều tiến trình trên các máy thực.

Tuy nhiên, nhược điểm lớn của giải pháp này là thời gian thực hiện của chương trình tăng chứ không là làm giảm như kỳ vọng. Điều này làm cho giải pháp không có ý nghĩa gì khi xét về mặt hiệu năng, tức là không có ý nghĩa gì cho việc nâng cao hiệu suất hoạt động của hệ thống.

2.3 Mô hình CAPE song song hai mức đa luồng[2]

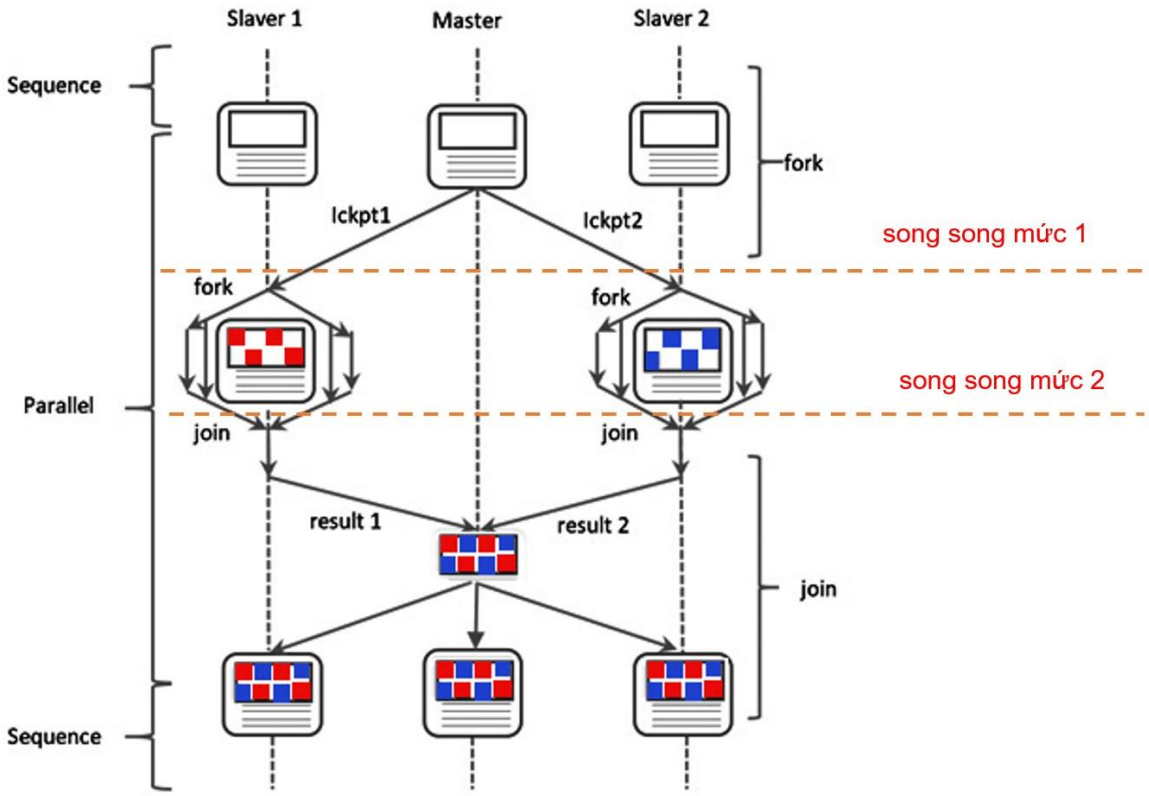
2.3.1 Ý tưởng thực hiện

Nguyên tắc căn bản của phương pháp này là cài đặt các cấu trúc song song của OpenMP qua 2 mức. Mức thứ nhất được thực hiện theo mô hình hiện tại của CAPE, tức là phân chia công việc cho các tiến trình trên các nút phụ và tổng hợp kết quả tính toán từ các nút này nhờ kỹ thuật chụp ảnh tiến trình. Mức thứ hai được thực hiện ngay chính trên các nút tính toán, đối với mỗi phần việc con của cấu trúc song song tiếp tục được song song hóa trên nhiều luồng để khai thác khả năng của các bộ vi xử lý đa lõi và từ đó tăng được tốc độ thực hiện.

Với yêu cầu mới song song hóa trên mỗi nút phụ, mỗi đoạn mã song song hóa của OpenMP phải được chia thành nhiều tiến trình (process) theo số nút tính toán và nhiều luồng (thread) trên mỗi nút tính toàn để chạy trên mỗi nút phụ đồng thời nhiều luồng để khai thác tốt hơn hiệu năng của CPU đa lõi. Hình 2.7 minh họa mô hình CAPE mới đáp ứng yêu cầu chạy nhiều luồng trên từng nút phụ.

Về mặt lý thuyết, đây chính là giải pháp có thể nâng cao nhiều nhất hiệu suất hoạt động của hệ thống, bằng cách song song hóa đoạn mã tính toán trên nút phụ theo mô hình CAPE song song hai mức đa luồng, mỗi luồng thực hiện một phần công việc của đoạn mã tính toán. Điều này có thể dễ dàng chứng minh theo luật Amdahl [36] về hệ số tăng tốc trên lý thuyết khi triển khai song song của một thuật toán trên nhiều đơn vị xử lý, do chương trình ứng dụng trong giải pháp này được song song hóa thành nhiều phần hơn so với trong mô hình của CAPE đơn luồng.

[2] Đa luồng (Multithreading) là khả năng xử lý nhiều luồng cùng lúc của CPU. Khi có nhiều nhiệm vụ khác nhau, CPU có thể làm việc một cách song song (parallel) nhiều luồng đối với CPU chỉ có một lõi hoặc làm việc đồng thời (concurrent) đối với CPU đa lõi. Lõi là nơi thực hiện các nhiệm vụ và mỗi lõi chỉ chạy 1 nhiệm vụ tại một thời điểm.



Hình 2.7. Mô hình hoạt động song song 2 mức của CAPE đa luồng

Trong mô hình mới này, mỗi phần song song OpenMP ban đầu cũng được chia thành nhiều phần việc con, mỗi phần thực thi trong một nút phụ như trong mô hình thực thi CAPE đơn luồng. Điểm mới là mỗi phần việc trên các nút phụ tiếp tục được phân thành nhiều luồng, thay vì luồng đơn như trong mô hình hiện tại. Do vậy, ứng dụng chạy tại nút phụ có thể khai thác tốt hơn hiệu năng của các hệ thống CPU đa lõi. Như vậy, với mô hình này CAPE sẽ thực thi song song 2 mức, mức 1 liên quan đến phân chia và chạy đồng thời các tác vụ trên nhiều máy, trong khi mức 2 liên quan đến việc sử dụng nhiều luồng trên mỗi nút phụ để chạy song song phần nhiệm vụ được chia của nó.

2.3.2 Khuôn mẫu dịch cấu trúc `omp parallel` trong CAPE đa luồng

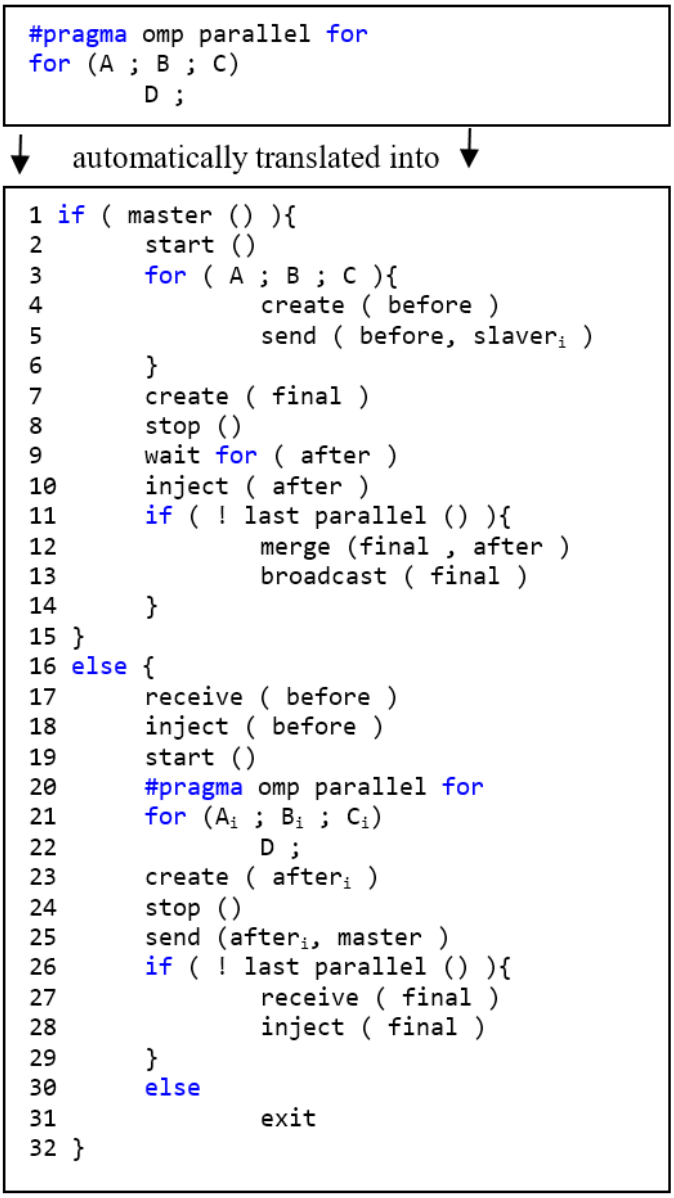
Như trong Hình 2.8, đoạn mã song song của OpenMP, theo mô hình CAPE đa luồng, được biên dịch thành một tập hợp các lệnh thuộc dạng ngôn ngữ C/C++ chứa cả chỉ thị OpenMP và CAPE. Điều này cung cấp khả năng chạy ở 2 mức độ song song, mức 1 phân chia công việc của vòng `for` ban đầu cho nhiều nút phụ và mức 2 sử dụng nhiều luồng để thực hiện song song hóa các công việc được giao trên mỗi nút. Chương

trình dịch C/C++ tiếp theo sẽ dịch tiếp mã này sang dạng mã máy chạy được trên nền tảng CAPE có hỗ trợ OpenMP tại các nút phụ.

Trong Hình 2.8, các mã dòng từ 2 đến 14 được thực thi ở nút chính, trong khi các dòng từ 17 đến 31 được chạy trên các nút phụ.

Trên nút chính, tại dòng 2, bộ nhớ của chương trình ứng dụng được đặt ở chế độ bảo vệ ghi và trình chụp ảnh tiến trình được khởi tạo. Các dòng 4-6 được sử dụng để tạo ảnh tiến trình và gửi ảnh chụp tiến trình đến các nút phụ để khởi tạo trạng thái giá trị ban đầu cho trình ứng dụng và phân nhiệm vụ cho trình ứng dụng trên nút phụ bằng cách tách chỉ số từ vòng lặp `for`. Sau đó, nút phụ tạo một ảnh chụp tiến trình để trích xuất kết quả thực hiện nhiệm vụ được phân chia và tạm dừng thực hiện giám sát chụp ảnh tiến trình. Ở các dòng 9-10, nút chính nhận kết quả thực hiện từ tất cả các nút phụ, sau đó tích hợp vào bộ nhớ của trình ứng dụng. Nếu có đoạn lệnh song song khác tiếp theo, nút chính sẽ hợp nhất các kết quả từ các nút phụ với kết quả cục bộ và phân phối ảnh chụp tiến trình tổng hợp tới tất cả các nút phụ để đồng bộ hóa không gian bộ nhớ của tiến trình trên tất cả các nút, chuẩn bị cho các thực hiện đoạn mã tiếp theo.

Trên các nút phụ, ở các dòng 17-19, mỗi nút nhận một ảnh chụp tiến trình của nút chính và tích hợp vào bộ nhớ của trình ứng dụng. Đây là bước nhận giá trị khởi tạo của chương trình và nhận thông tin phần việc được phân chia cho chương trình chạy trên nút phụ. Sau đó, tính năng giám sát để chụp ảnh tiến trình được bật để thực hiện đoạn mã song song của OpenMP – dòng 20-22. Đây là đoạn mã song song như chương trình gốc của OpenMP và được tổ chức chạy thành nhiều luồng như chương trình gốc OpenMP. Điểm khác biệt là số vòng lặp được điều chỉnh bằng thương số của số lần lặp trong chỉ thị `omp parallel for` ban đầu và số lượng nút phụ. Dòng 23-25 để trích xuất kết quả của chương trình tại nút phụ và gửi về nút chính. Sau đó, nút phụ nhận ảnh chụp tiến trình mới từ nút chính và đưa vào không gian bộ nhớ để chuẩn bị các phân đoạn mã song song tiếp theo, tương ứng các dòng 26-29.



Hình 2.8. Khuôn mẫu dịch cấu trúc `parallel for` trong CAPE đa luồng

2.4 Phân tích hiệu năng hoạt động của CAPE đa luồng

2.4.1 Hệ số tăng tốc lý thuyết theo luật Amdahl

Trước hết, chúng ta thực hiện việc phân tích một cách lý thuyết để đánh giá hiệu năng của CAPE sử dụng mô hình thực hiện song song 2 mức, cụ thể là đánh giá khả năng tăng tốc của CAPE khi thực hiện cùng một chương trình OpenMP ban đầu trên cùng một nền tảng. Giả sử chương trình này chứa các đoạn mã tuần tự và một chỉ thị `omp parallel for`. Các trường hợp chứa nhiều phần song song hơn có thể được phát triển tương tự.

Để làm điều này, chúng ta thực hiện việc so sánh mã nguồn ở 2 phiên bản, dạng CAPE sử dụng tiến trình đơn luồng – được trình bày tại Hình 1.8 – và dạng CAPE sử dụng tiến trình đa luồng – được trình bày tại Hình 2.8.

So sánh 2 đoạn mã trên, dễ dàng nhận thấy khác biệt ở câu lệnh số 20 (của khuôn dạng đơn luồng) và 20-22 (của khuôn dạng đa luồng), chính là việc thực hiện phần công việc được giao cho mỗi nút phụ ở dạng tiến trình đa luồng OpenMP thay cho một tiến trình đơn luồng.

Theo luật Amdahl [36], hệ số tăng tốc lý thuyết toàn cục của CAPE đa luồng so với CAPE đơn luồng được tính theo công thức sau:

$$S_{\text{latency}} = \frac{1}{(1-p) + \frac{p}{s_1 * s_2}} \qquad (1)$$

Trong đó

S_{latency} hệ số tăng tốc lý thuyết toàn cục

p tỉ lệ có thể song song hóa của thuật toán tuần tự

s_1 là số lần song song cấp 1, phân chia công việc trên bao nhiêu máy tính

s_2 là số lần song song cấp 2, trên từng máy tính được song song hóa thành bao nhiêu luồng/tiến trình.

Theo mô hình CAPE Hình 2.7, song song 2 mức, giả sử trường hợp chạy trên 8 máy mỗi máy sử dụng CPU 2 lõi. Giả thiết, đoạn code được chạy song song hóa toàn bộ khi đó p = 1 và thời gian giao tiếp dữ liệu giữa các máy được bỏ qua. Khi đó hệ số tăng tốc lý thuyết theo công thức (1) với trường hợp chạy 1 lõi CPU và 2 lõi CPU lần lượt như sau:

Khi đó hệ số tăng tốc lý thuyết của hệ thống với trường hợp chạy 1 lõi trên mỗi máy tính là

$$S_{\text{latency}} = \frac{1}{(1-1) + \frac{1}{8}} = 8$$

Trong trường hợp sử dụng 2 lõi

$$S_{\text{latency}} = \frac{1}{(1-1) + \frac{1}{8*2}} = 16$$

Trong thực tế, một chương trình OpenMP nguyên thuỷ luôn có các đoạn mã tuần tự. Bên cạnh đó, các câu lệnh CAPE để thực hiện việc phân chia công việc, trích rút kết quả, đồng bộ hoá bộ nhớ,… cũng tiêu tốn một chi phí nhất định. Do đó, hệ số tăng tốc ở trên chỉ mang tính lý thuyết (cận trên), thực tế luôn phải thấp hơn mức cận trên này và tiến tới giá trị gần với cận trên này khi thời gian thực hiện việc tính toán trên các nút tính toán phải lớn hơn rất nhiều so với thời gian thực hiện các công việc khác như chia việc, đồng bộ dữ liệu,….

Như vậy, trong trường hợp bỏ qua thời gian thực hiện các đoạn mã tuần tự và các chi phí khác, công thức trên trở thành:

$S_{latency} \sim s$

Tức là hệ số tăng tốc sẽ tỷ lệ thuận với s, là số luồng song song hoá, tương đương với số lõi của CPU tại các nút slave.

Đây cũng chính là giả thuyết khoa học chính của việc mở rộng mô hình hoạt động của CAPE. Giả thuyết này sẽ được kiểm chứng bởi các kết quả thực nghiệm ở phần sau.

2.4.2 Kết quả thực nghiệm và đánh giá

2.4.2.1 Hệ thống thực nghiệm và bài toán thực nghiệm

Để đánh giá bằng thực nghiệm hiệu năng của CAPE đa luồng, như được phân tích hệ số tăng tốc theo khía cạnh toán học trong mục 2.4.1, một số thực nghiệm chạy CAPE đa luồng đã được tiến hành trên hệ thống các máy tính cá nhân được kết nối bởi mạng LAN 100 Mbps [3] và được vận hành bởi hệ điều hành Ubuntu 18.04.3 LTS, OpenSSH-Server. Các thử nghiệm được tiến hành trên 2 cấu hình cluster như sau:

Hệ thống 1: Sử dụng 1 cluster gồm 16 nút slave có cấu hình như sau: 6 máy tính Intel(R) Core(TM) i3-2100 @3.1GHz RAM 8GB; 2 máy AMD Phenom(TM) II X4 925 Processor @2.80GHz, 8GB of RAM; 1 máy Intel(R) Core(TM) i3-2120 CPU @3.30GHz, RAM 4GB; 2 máy Intel(R) Pentium(R) Dual CPU E2160 @1.80GHz, 2GB of RAM; 5 máy Intel(R) Core(TM)2 Duo CPU E7300 @2.66GHz, 3G of RAM. Với cấu hình này chỉ chạy được 2 kịch bản sử dụng 1 lõi và 2 lõi trên mỗi máy, do giới hạn của các máy sử dụng CPU Duo core (2 lõi).

[3] Các máy tính cùng lớp mạng được nối chung cùng 1 switch 48 ports 10/100 Mbps, dây mạng CAT5.

Hệ thống 2: Sử dụng 1 cluster gồm 8 nút slave có cấu hình như sau: 6 máy Intel(R) Core(TM) i3-2100 @3.1GHz RAM 8GB; 2 máy AMD Phenom(TM) II X4 925 Processor @2.80GHz, 8GB of RAM. Với cấu hình này có thể chạy được 4 kịch bản sử dụng 1 đến 4 lõi trên mỗi máy.

Các thực nghiệm đã tiến hành chạy chương trình nhân hai ma trận vuông có các kích thước lần lượt là 9600×9600 và 6400×6400, với số luồng tại các nút slave biến thiên từ 1 đến 2 trên hệ thống 1; và 1 đến 4 luồng trên hệ thống 2. Mỗi trường hợp đều được chạy 10 lần và kết quả cuối cùng là giá trị trung bình cộng của 10 lần chạy. Đặc tính của bài toán cùng với kích thước dữ liệu này thoả mãn điều kiện về thời gian tính toán tại các nút slave lớn hơn nhiều so với thời gian thực hiện các công việc khác của chương trình, như đã đề cập ở mục 2.4.1.

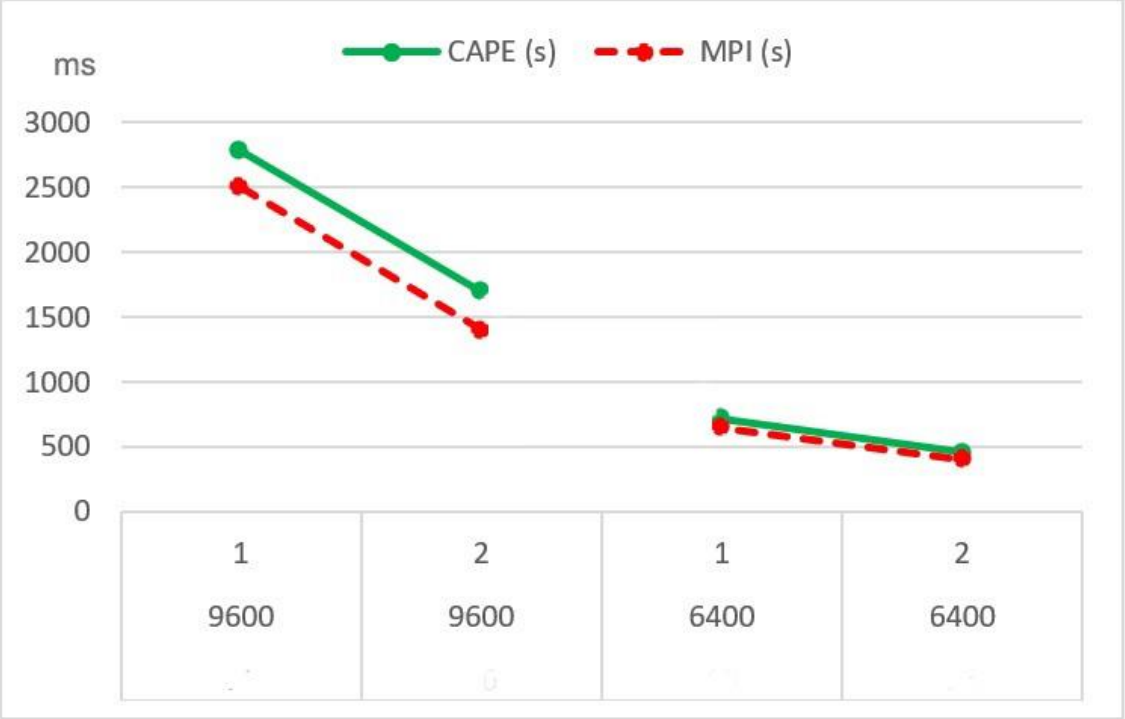
Để đánh giá hiệu năng hệ thống khi áp dụng mô hình CAPE mới, đồng thời làm nổi bật ưu điểm của nó, chúng tôi tiến hành hai hướng so sánh lớn : 1) So sánh với CAPE sử dụng mô hình hoạt động trước đây (đơn luồng) ; và 2) So sánh với MPI. Hướng thứ nhất sẽ tính được chính xác hệ số tăng tốc của mô hình mới, theo công thức tính của luật Amdahl. Trong khi hướng thứ hai – so sánh với MPI là phương pháp có khả năng cung cấp hiệu năng cao nhất cho lập trình song song trên hệ thống sử dụng bộ nhớ phân tán – dùng để đánh giá hiệu năng của CAPE so với các phương pháp khác. Cũng cần nhắc lại rằng CAPE, với những chi phí thời gian và tài nguyên mang tính tất yếu từ kỹ thuật cơ sở của CAPE, sẽ không thể có hiệu năng sánh bằng MPI. Điều này cũng đúng với tất cả các kỹ thuật lập trình song song trên hệ thống sử dụng bộ nhớ phân tán khác. Ưu điểm thúc đẩy CAPE và những kỹ thuật khác nằm ở khả năng cung cấp giải pháp lập trình đơn giản hơn, ít tốn thời gian và công sức hơn cho người lập trình hơn so với MPI.

2.4.2.2 Kết quả thực nghiệm và phân tích, đánh giá

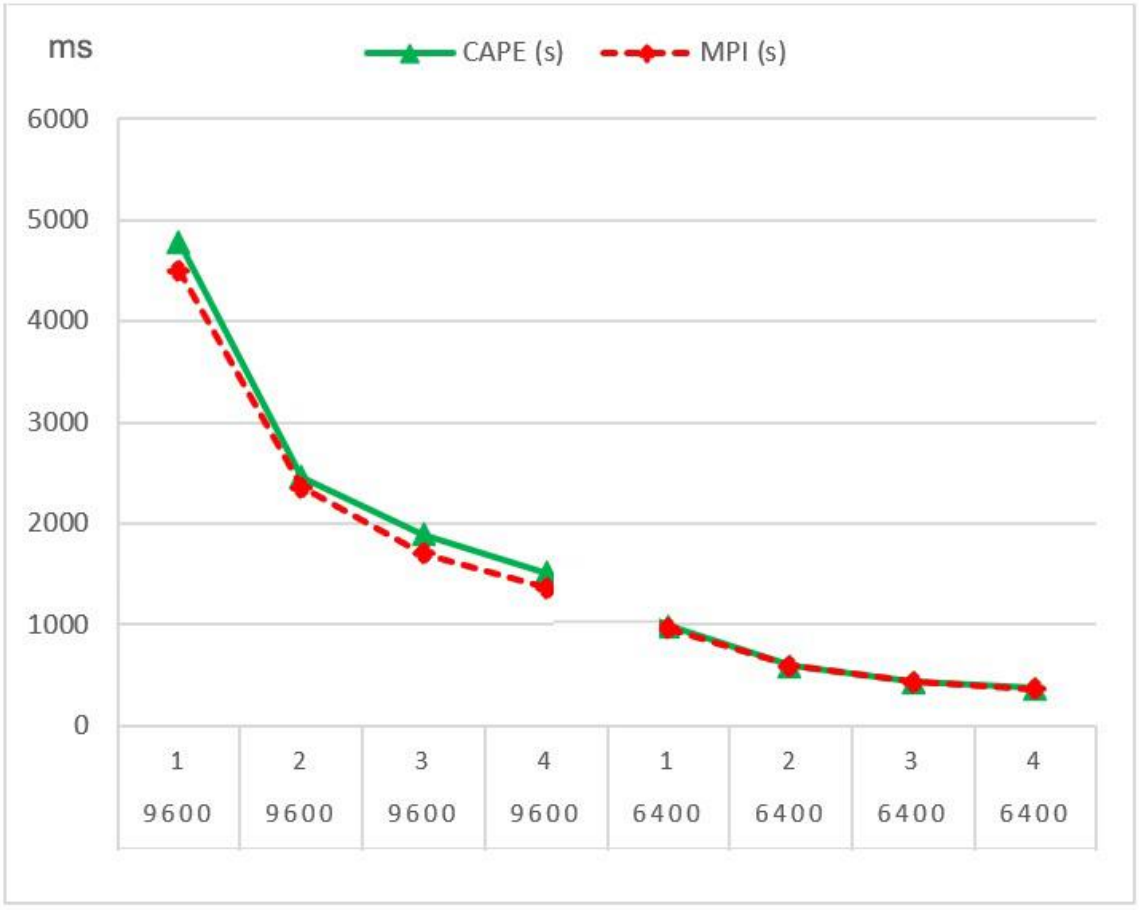
Hình 2.9 và Hình 2.10 trình bày kết quả chạy thời gian chạy cùng 1 chương trình theo phương pháp CAPE và MPI, biến thiên theo số lượng lõi (tương đương số lượng luồng/tiến trình) sử dụng trên từng máy và kích thước bài toán (ma trận 9600x9600 hoặc 6400x6400). Trong đó, Hình 2.9 tương ứng với Hệ thống 1 (16 nút phụ), chỉ thử nghiệm với trường hợp sử dụng 1 lõi/luồng và 2 lõi/luồng; Hình 2.10 tương ứng với Hệ thống 2 (8 nút phụ), thử nghiệm với số lõi/luồng từ 1 đến 4. Trên cả hai biểu đồ, trục

tung chỉ thời gian thực hiện chương trình với đơn vị là milliseconds; Trục hoành có 2 chỉ số: kích thước ma trận (9600x9600 hoặc 6400x6400); và số luồng sử dụng (1 hoặc 2 hoặc 3 hoặc 4), trong đó 1 là trường hợp sử dụng một lõi, tương ứng với sử dụng phiên bản CAPE đơn luồng; và 2/3/4 tương ứng với phiên bản CAPE đa luồng và sử dụng 2/3/4 lõi CPU trên các nút phụ.

2.4.2.2.1 Kết quả và đánh giá theo thời gian thực hiện



Hình 2.9. Thời gian thực hiện trên cluster 16 máy biến thiên theo số lõi sử dụng



Hình 2.10. Thời gian thực hiện trên trên cluster 8 máy biến thiên theo số lõi sử dụng

So sánh giữa CAPE đa luồng và CAPE đơn luồng

Đối với cả 2 hệ thống và 2 kích thước bài toán trên đó (9600x9600 và 6400x6400), thời gian thực hiện của CAPE đa luồng luôn bé hơn của CAPE đơn luồng. Điều này chứng minh một cách rõ ràng mô hình hoạt động đa luồng đã mang lại hiệu năng thực hiện tốt hơn mô hình đơn luồng trước đây.

Với hệ thống cluster 8 máy, biểu đồ ở Hình 2.10 cho thấy độ dốc tăng tốc từ chuyển đổi 1 lõi thành 2 lõi luôn lớn hơn độ dốc khi tăng từ 2 lõi lên 3 lõi, cũng như từ 3 lõi lên 4 lõi. Điều này có thể là do cấu trúc CPU Intel(R) Core(TM) i3, tuy được xem là có 4 lõi nhưng thực chất chỉ có 2 lõi vật lý và mỗi lõi chứa 2 siêu phân luồng.

So sánh giữa CAPE và MPI

Trong tất cả các trường hợp, thời gian chạy chương trình theo MPI luôn nhanh hơn hơn CAPE, nhưng với một tỷ lệ tương đối ổn định, thể hiện qua đồ thị đường kẻ thời gian chạygần như song song với nhau. Điều này cũng chứng tỏ CAPE vận hành ổn định trong trường hợp tăng hoặc giảm số lõi CPU sử dụng. Hơn nữa, mức chênh lệch thời gian chạy chương trình của CAPE và MPI chỉ ở mức xấp xỉ 8%. Mức chênh lệch

này là không cao và có nguyên nhân từ việc CAPE luôn tốn chi phí thời gian để thực hiện việc giám sát và chụp ảnh tiến trình. Với mức chênh lệch này, có thể nói CAPE đã tiệm cận được với phương pháp lập trình song song trên hệ thống sử dụng bộ nhớ phân tán có khả năng cung cấp hiệu năng cao nhất là MPI. Kết hợp với đặc tính nổi bật của OpenMP là tính dễ học, dễ sử dụng, tiết kiệm công sức và thời gian lập trình, có thể nói CAPE có khả năng cung cấp một cách thức lập trình song song ưu việt trên hệ thống sử dụng bộ nhớ phân tán.

2.4.2.2.2 Kết quả và đánh giá theo hệ số tăng tốc

Để thấy rõ hơn tính hiệu quả của mô hình mới, ta tính toán hệ số tăng tốc của nó so với mô hình cũ. Hệ số tăng tốc thực tế được đánh giá theo công thức:

$$S_{latency} = \frac{T_{\text{đơn luồng}}}{T_{\text{đa luồng}}}$$

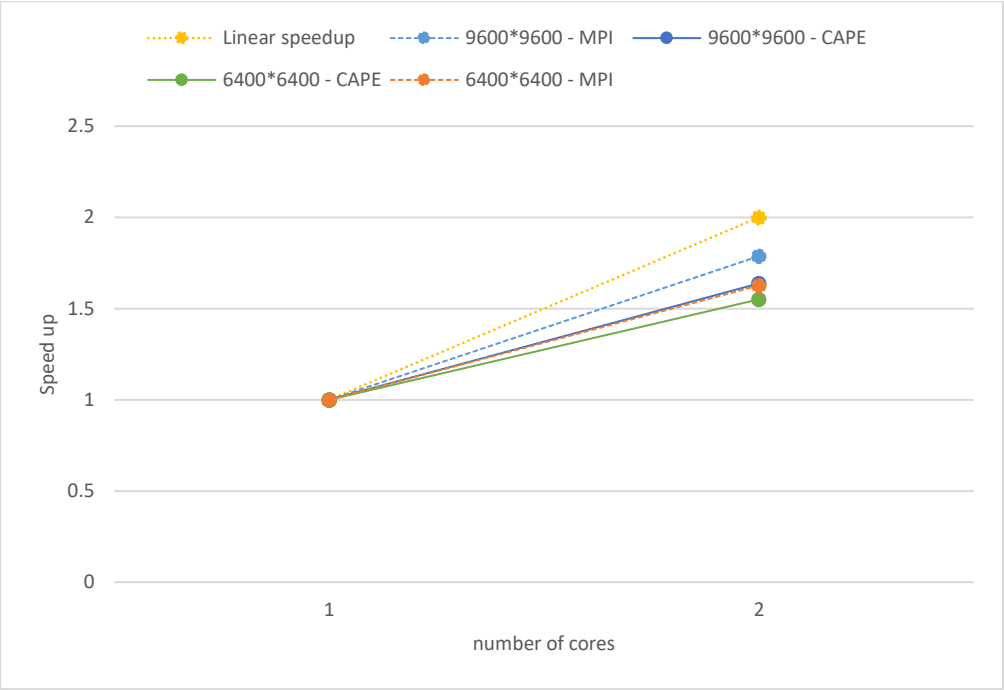
Trong đó:

$S_{latency}$ là hệ số tăng tốc;

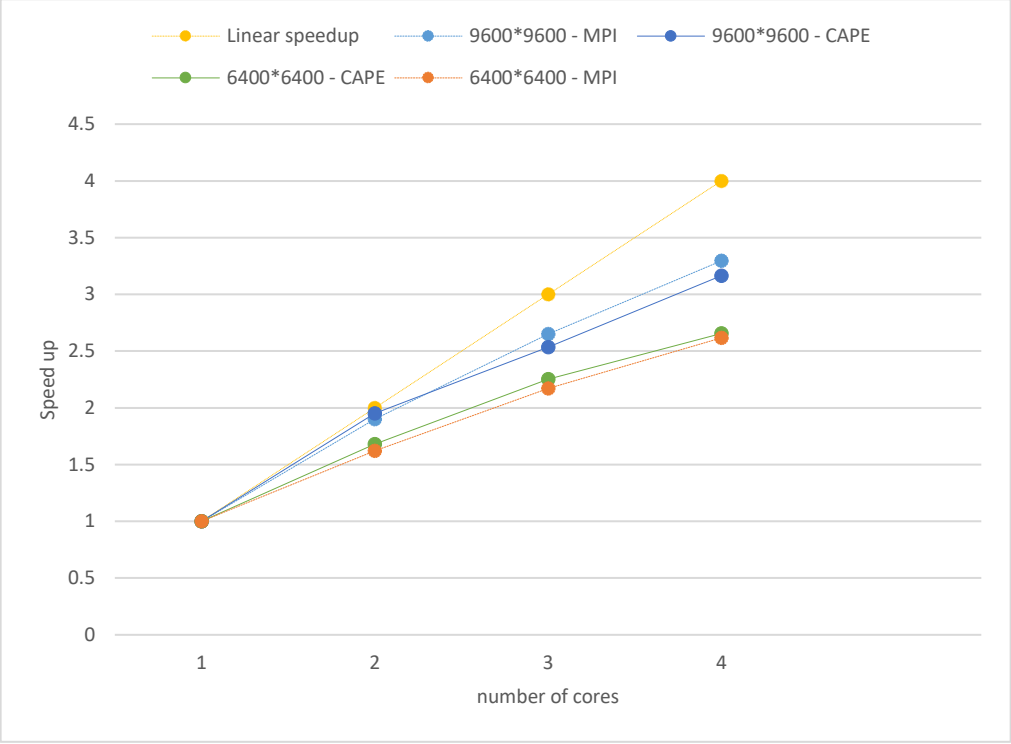
$T_{đơn\ luồng}$ là thời gian thực hiện chương trình với mô hình CAPE đơn luồng, tương ứng với trường hợp sử dụng 1 lõi/luồng ở các Hình 2.9 và Hình 2.10;

$T_{đa\ luồng}$ là thời gian thực hiện chương trình với mô hình CAPE đa luồng, tương ứng với trường hợp sử dụng 2-4 lõi/luồng ở các Hình 2.9 và Hình 2.10.

Kết quả tính toán được biểu diễn trên các Hình 2.11 và Hình 2.12, trong đó bao gồm cả biểu diễn hệ số tăng tốc của CAPE đa luồng so với CAPE đơn luồng và hệ số tăng tốc của MPI sử dụng 1 tiến trình ở các nút phụ và đa tiến trình ở cácnút phụ. Trên các biểu đồ này, trục tung biểu diễn hệ số tăng tốc (lần), trục hoành tương ứng với số lõi/luồng được sử dụng.



Hình 2.11. Hệ số tăng tốc theo số lõi CPU sử dụng trên hệ thống 16 nút phụ



Hình 2.12. Hệ số tăng tốc theo số lõi CPU sử dụng trên hệ thống 8 nút phụ

So sánh giữa CAPE đa luồng và CAPE đơn luồng

Trong cả hai hình trên, đồ thị của CAPE đơn luồng chỉ có 1 điểm duy nhất tương ứng với trường hợp sử dụng 1 lõi. Đối sánh với các trường hợp CAPE đa luồng sử dụng 2 đến 4 lõi, ta thấy trên đồ thị gần như tạo thành 1 đường tăng tốc tăng tuyến tính. Một cách chính xác, tốc độ tăng tốc của CAPE và MPI cao hơn trong trường hợp sử dụng 2

lõi là 1,90-1,95 lần khi so sánh với trường hợp sử dụng 1 lõi. Đồng thời khi sử dụng 4 lõi của CPU (thực chất là 2 lõi thực và 2 siêu phân luồng), kết quả thực nghiệm hệ số tăng tốc vẫn cao, có thể lên tới 3.16 so với trường hợp sử dụng 1 lõi CPU. Hơn nữa, đường tăng tốc này gần như tiệm cận với đường tăng tốc lý thuyết lý tưởng ở trên cùng, chứng tỏ hệ số tăng tốc này là rất tốt.

Những kết luận này đã nghiệm được giả thiết đề ra, được nêu ở mục trên là:

$S_{latency} \sim s$

Tức là hệ số tăng tốc sẽ tỷ lệ thuận với s, là số luồng song song hoá, tương đương với số lõi của CPU tại các nút phụ.

So sánh giữa CAPE và MPI

Các đường gấp khúc của CAPE gần như gần sát với các đường của MPI thể hiện CAPE có hệ số tăng tốc ổn định và xấp xỉ với MPI. Khi kích thước ma trận nhỏ hơn hệ số tăng tốc thấp hơn do thời gian phân chia công việc và đồng bộ hóa dữ liệu chiếm một tỷ lệ cao hơn so với thời gian tính toán.

Lưu ý lại một lần nữa, MPI là phương pháp cung cấp hiệu năng cao nhất cho việc lập trình song song trên hệ thống sử dụng bộ nhớ phân tán. Do đó hệ số tăng tốc của CAPE như trên đã là rất tốt.

2.5 Tiểu kết Chương II

Chương này trình bày trọng tâm chính của đề tài, là đề xuất mô hình hoạt động mới của CAPE theo kiến trúc 2 mức song song, trong đó mức 2 sử dụng cấu trúc đa luồng của OpenMP. Phiên bản CAPE sử dụng mô hình hoạt động này được gọi là CAPE đa luồng.

Các kết quả đánh giá hiệu năng của CAPE đa luồng, cả về lý thuyết cũng như thực nghiệm đã chứng minh mô hình hoạt động mới này có khả năng gia tăng hệ số tăng tốc rất tốt, tỷ lệ thuận với số lõi/luồng được sử dụng tại các nút tính toán và chênh lệch khá ít với hệ số của MPI – phương pháp cung cấp hiệu năng tốt nhất cho lập trình song song trên các hệ thống bộ nhớ phân tán. Điều này chứng tỏ mô hình hoạt động mới hoạt động hiệu quả hơn nhiều so với mô hình cũ, CAPE đơn luồng, khi sử dụng CAPE trên các hệ thống máy tính có CPU đa lõi.

Các kết quả của chương này được công bố ở công trình [CT2] [CT5].

CHƯƠNG 3 KỸ THUẬT CHỤP ẢNH TIẾN TRÌNH CHO CAPE ĐA LUỒNG

Như đã giới thiệu ở phần mở đầu, CAPE đặt cơ sở trên kỹ thuật chụp ảnh tiến trình, với 2 phiên bản, tương ứng với 2 kỹ thuật chụp ảnh tiến trình là Kỹ thuật chụp ảnh tiến trình đầy đủ và Kỹ thuật chụp ảnh tiến trình gia tăng rời rạc. Khi chuyển qua mô hình hoạt động song song hai cấp đa luồng, công cụ chụp ảnh tiến trình gia tăng rời rạc đã được phát triển trước đó cho CAPE đơn luồng không còn sử dụng được nữa. Do đó, cần phải phải phát triển một phiên bản chụp ảnh tiến trình mới. Mặt khác, phiên bản chụp ảnh tiến trình và CAPE phiên bản cũ được phát hành trên hệ điều hành Ubuntu 10.04. Do đó, việc nâng cấp lần này, để chạy được trên hệ điều hành mới hơn lại càng phức tạp và đòi hỏi phải nghiên cứu lại từ những vấn đề căn bản nhất của Kỹ thuật chụp ảnh tiến trình.

3.1 Kỹ thuật chụp ảnh tiến trình

3.1.1 Khái niệm Kỹ thuật chụp ảnh tiến trình

Chụp ảnh tiến trình (Checkpointing Technique) là một kỹ thuật cơ bản trong việc phục hồi trạng thái chương trình ở các thời điểm chạy trước đó. Khái niệm và nguyên lý cơ bản của Kỹ thuật chụp ảnh tiến trình đã được trình bày trong phần giới thiệu về CAPE đơn luồng, trang 33.

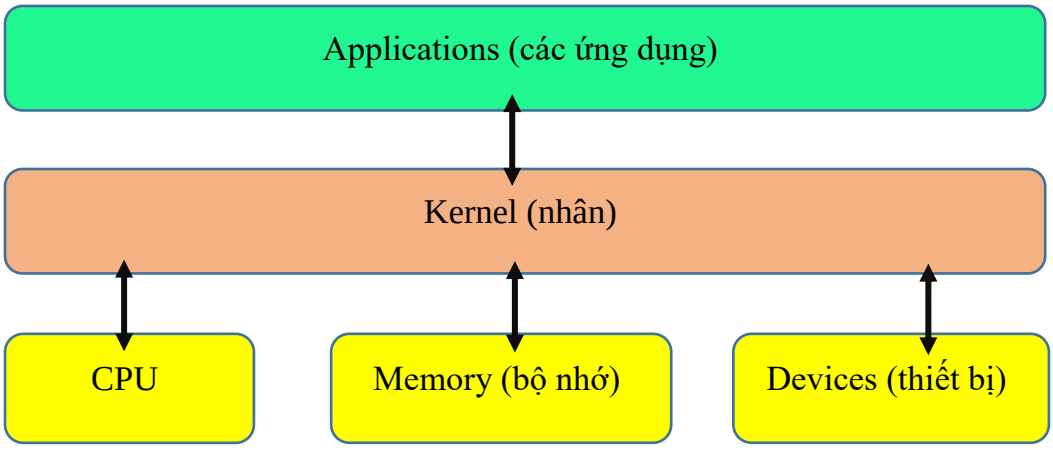
3.1.2 Không gian nhân và không gian (Kernel mode) người sử dụng (User mode)

Hiện tại, CAPE đang được phát triển trên Ubuntu, một phiên bản rất phổ biến của hệ điều hành Linux. Do đó, các kiến thức liên quan trong các phần còn lại của chương này được giới hạn trên hệ điều hành Linux.

Phần nhân (kernel) của Linux được phân phối theo giấy phép bản quyền GNU General Public License version 2 (GPLv2) [72]. Linux sử dụng cơ chế hoạt động nhân nguyên khối (monolithic kernel architecture) với tất cả các hoạt động của hệ thống được làm việc trong không gian nhân (kernel) [73]. Nhân (kernel) chứa một tập các hàm khởi tạo và hàm gọi hệ thống để thực hiện tất cả các dịch vụ của hệ điều hành như: Quản lý các tiến trình, quản lý bộ nhớ, quản lý thiết bị, giải quyết tranh chấp tài nguyên,… Các chương trình điều khiển thiết bị chuyên biệt (driver) có thể được thêm vào nhân của Linux như là những mô-đun bổ sung.

Không gian người sử dụng (user mode) bao gồm tất cả chương trình bên ngoài phần mã của nhân hệ điều hành. Các chương trình ứng dụng hoặc các thư viện thực hiện trên nền hệ điều hành thông qua việc tương tác với nhân của nó. Mỗi chương trình ứng dụng chạy ở không gian người sử dụng được hệ điều hành cấp cho một vùng nhớ ảo và một cách tổng quan một chương trình ứng dụng này không thể truy cập vào vùng nhớ ảo của một chương trình khác. Và do các chương trình ở không gian người sử dụng có không gian địa chỉ riêng biệt, nếu một chương trình bị lỗi, chỉ mình nó bị ảnh hưởng mà không làm hỏng các chương trình khác hoặc toàn bộ hệ điều hành.

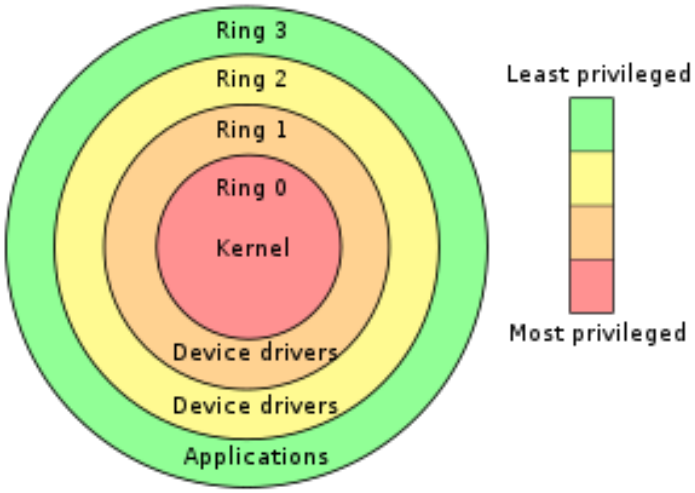
Một cách khác để mô tả các ứng dụng ở không gian người sử dụng là sử dụng thuật ngữ "ít đặc quyền" (less privileged). Hệ điều hành hạn chế các ứng dụng ở không gian người sử dụng truy cập trực tiếp vào các tài nguyên hệ thống quan trọng, do đó làm cho trình ứng dụng ít có đặc quyền hơn. Ví dụ, nếu một ứng dụng muốn truy cập vào phần cứng, ứng dụng này phải thông qua nhân của hệ điều hành bằng cách sử dụng lệnh gọi hệ thống (system calls).



Hình 3.1. Minh họa việc gọi truy cập/sử dụng các tài nguyên phần ứng của các trình ứng dụng

Phần nhân (Kernel) là thành phần cốt lõi mà tất cả các thành phần khác bên trong hệ điều hành phải dựa vào. Kernel quản lý phần cứng máy tính, lập lịch cho các quy trình chạy trên máy tính cũng như xử lý các tương tác giữa phần cứng và phần mềm ứng dụng. Tóm lại, kernel là đoạn mã đặc quyền nhất chạy trên hệ thống, bởi nó là mã tương tác trực tiếp với phần cứng. Mọi chương trình khác muốn sử dụng tài nguyên phần cứng phải yêu cầu quyền truy cập thông qua kernel.

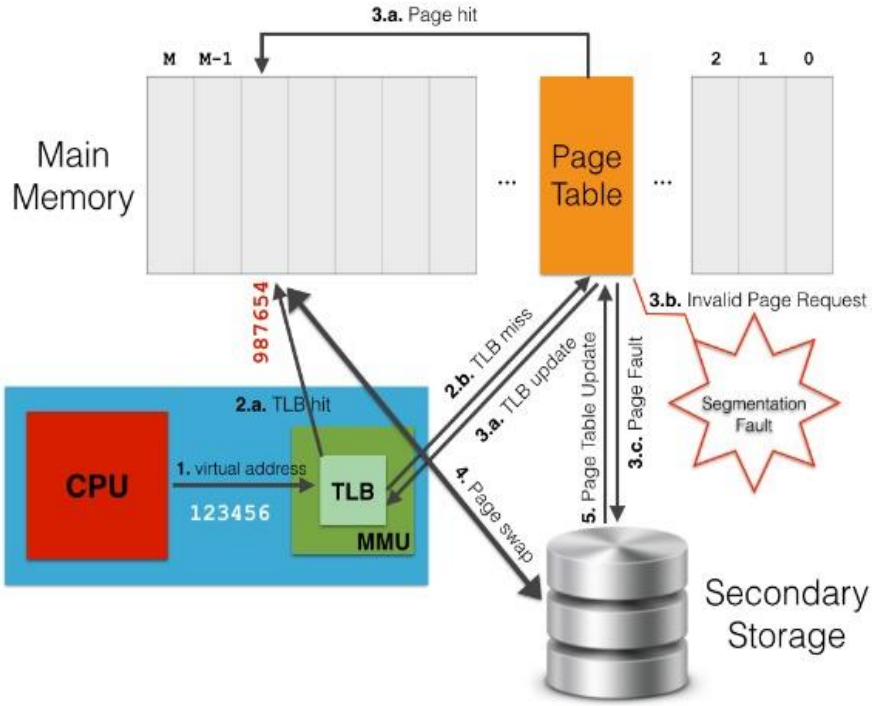
Hầu hết các kiến trúc CPU (x86 và x86-64) và hệ điều hành đều sử dụng cơ chế phân thành nhiều tầng đặc quyền (privilege), thường gọi vòng đặc quyền (privilege rings). Kiến trúc x86 và x86-64 có 4 đặc quyền được đánh số từ 0 đến 3, được minh họa trên Hình 3.2. Linux cũng sử dụng cơ chế vòng đặc quyền, nhưng chỉ sử dụng 2 vòng 0 và 3 cho mức giám sát (supervisory) và mức người sử dụng (user). Như vậy để yêu cầu hoạt động I/O (xuất/nhập) hoặc dịch vụ quản lý vùng nhớ, các trình ứng dụng phải gọi hàm hệ thống do nhân cung cấp, trình ứng dụng không đủ đặc quyền để trực tiếp quản lý những tài nguyên này.



Hình 3.2. Minh họa các vòng đặc quyền trong hệ điều hành

3.1.3 Cơ chế cấp phát bộ nhớ ảo cho trình ứng dụng trên Linux

Khi một chương trình được chạy, nó được hệ điều hành tải vào vùng nhớ do hệ điều hành cấp phát và trở thành một tiến trình mới. Linux quản lý bộ nhớ theo trang, đối với kiến trúc x86 và x86-64 mỗi trang có kích thước 4KB. Một bảng danh mục trang (Page table) được sử dụng để quản lý các trang nhớ được cấp phát. Trên hệ điều hành 32 bit, mỗi chương trình khi chạy được hệ điều hành cấp 4GB vùng nhớ ảo, trong đó có 1GB cho không gian nhân và 3GB cho không gian người sử dụng. Vùng nhớ ảo này được tổ chức thành một dãy các trang nhớ liên tục, minh họa trong Hình 3.3.



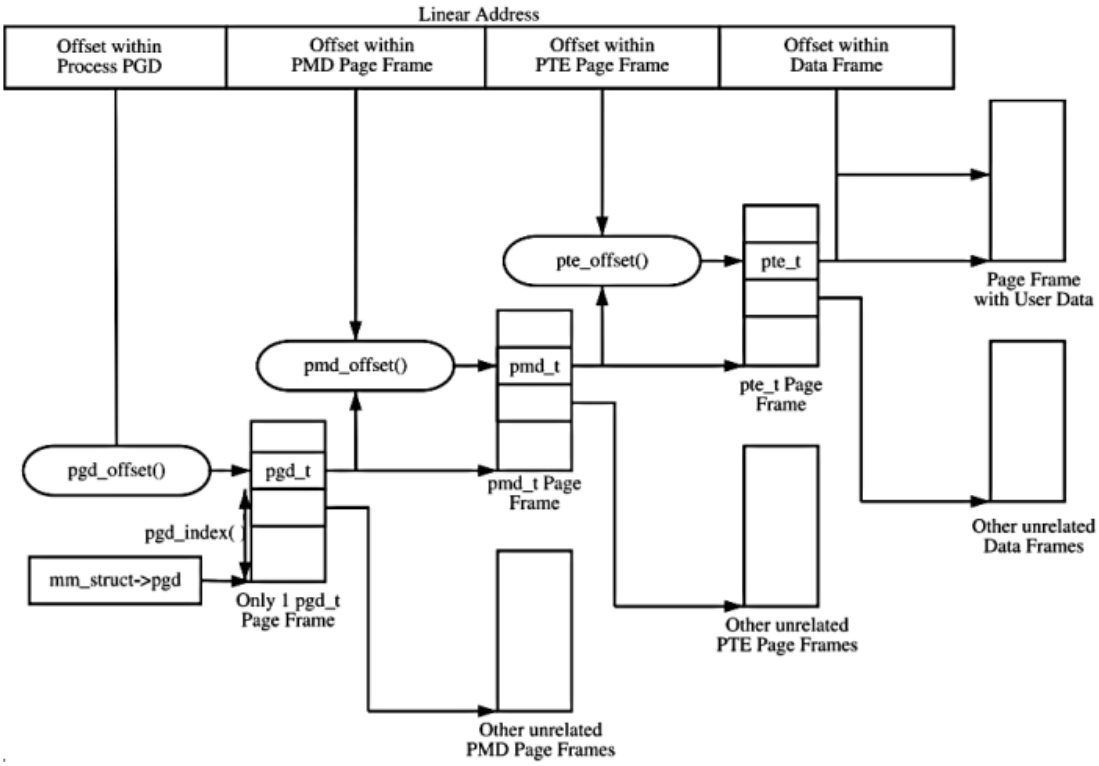
Hình 3.3. Ánh xạ địa chỉ bộ nhớ ảo sang địa chỉ vật lý

Quá trình ánh xạ một địa chỉ ảo sang một địa chỉ vật lý được minh họa trên Hình 3.3 như sau:

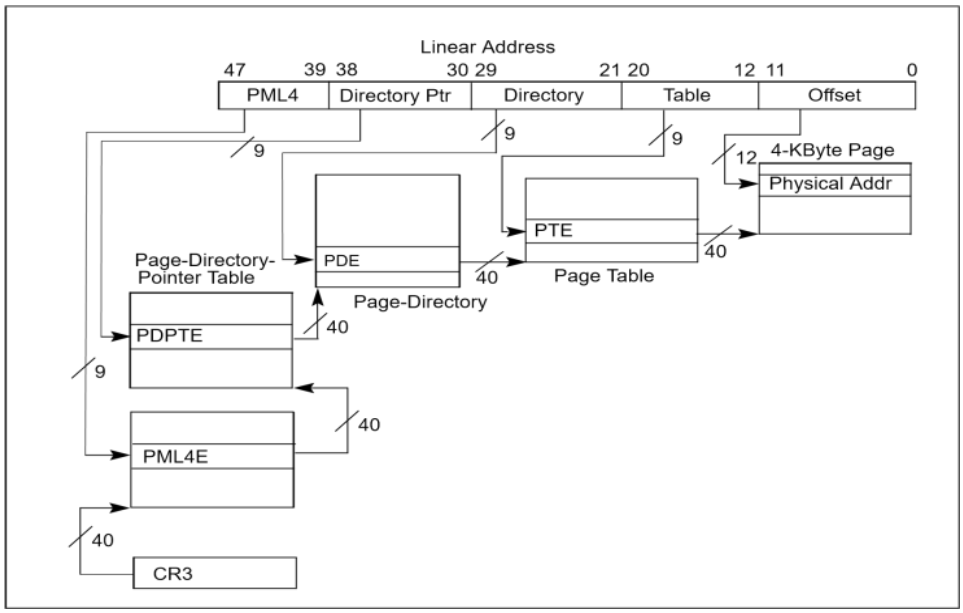
Bước 1: MMU (Memory Management Unit – Bộ quản lý bộ nhớ) sẽ tìm địa chỉ này trong TLB (Translation Lookaside Buffer). Nếu tìm thấy sẽ tiếp tục Bước 2.a - thực hiện đọc/ghi dữ liệu. Nếu không, MMU sẽ tìm trong toàn bộ Page table - tương ứng với Bước 2.b. Nếu tìm thấy, địa chỉ sẽ được cập nhật lại vào TLB (Bước 3.a). Quá trình tiếp tục theo Bước 1 và Bước 2.a.

Bước 2.b - địa chỉ vùng nhớ không tìm thấy: Có thể xảy ra 2 tình huống. 1) Địa chỉ không tồn tại, và 2) trang nhớ không được tải trong vùng nhớ. Với trường hợp 1 một lỗi (exception) được phát sinh (Bước 3.b). Với trường hợp 2 tương ứng với Bước 3.c - page fault, có nghĩa là trang nhớ được yêu cầu nằm ở vùng nhớ thứ cấp (secondary storage). Trình quản lý bộ nhớ của hệ điều hành cập nhật thông tin page fault (Bước 4) và cập nhật Page table (Bước 5). TLB thực hiện việc cập nhật giữa địa chỉ vùng nhớ ảo và vùng nhớ vật lý nơi trang nhớ vừa được tìm thấy (Bước 3.a), sau đó hoạt động tiếp tục với Bước 1 và Bước 2.a.

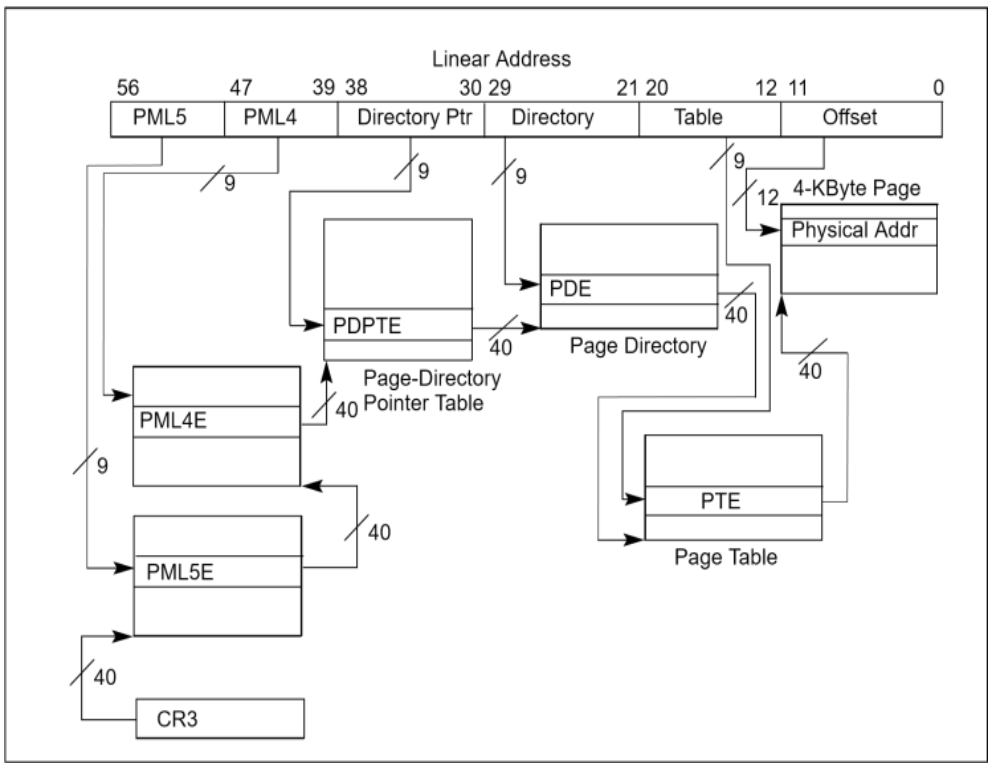
Từ kernel 2.6.10, page table hỗ trợ 4 cấp, tăng 1 cấp so với phiên bản trước đó. Page table 5 cấp được hỗ trợ từ kernel 4.11-rc2 100[45]. Hình 3.4 minh họa cấu trúc 3 cấp page table [46].



Hình 3.4. Cấu trúc tổ chức phân trang bộ nhớ 3 cấp [46]



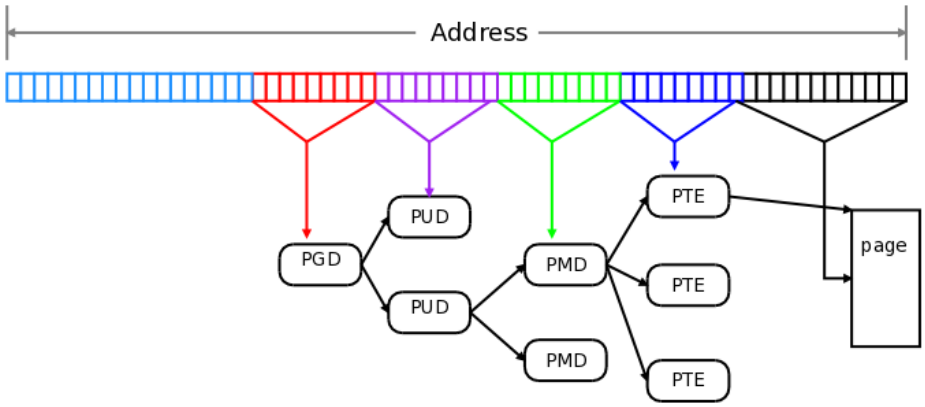
Hình 3.5. Sơ đồ tổ chức địa chỉ bộ nhớ phân trang 4 cấp của CPU Intel [47]



Hình 3.6. Sơ đồ tổ chức địa chỉ bộ nhớ phân trang 5 cấp của CPU Intel [47]

Với tổ chức trang nhớ 4 cấp, vùng địa chỉ của bộ nhớ ảo được sử dụng là 48/64 bits (từ bits 0-47), chi tiết phân đoạn từng vùng bits được trình bày trong Hình 3.5 [47]. Với tổ chức trang nhớ 5 cấp, địa chỉ vùng nhớ ảo có thể tổ chức sử dụng thêm 9 bits từ 48-56 [47] – như minh họa ở Hình 3.6 – giúp cho việc tăng kích thước vùng nhớ ảo có thể quản lý từ 128 TB đến 128 PiB.

Tương ứng với sự hỗ trợ của quản lý bộ nhớ ảo 5 cấp của phần cứng, Linux tích hợp hỗ trợ quản lý bộ nhớ ảo 5 cấp từ kernel 4.11-rc2.



Hình 3.7. Minh họa cấu trúc trang bộ nhớ ảo 5 cấp trong Linux

Hình 3.7 [48] minh họa tổ chức trang nhớ 5 cấp, với đối tượng PUD được bổ sung thêm vào giữa 2 mức PMD và PGD để tương thích với hỗ trợ tổ chức vùng nhớ ảo 5

cấp của phần cứng. Chi tiết tổ chức phân đoạn bit mỗi cấp được giới thiệu trong [48][49].

3.1.4 Phân loại kỹ thuật chụp ảnh tiến trình

Có nhiều cách phân loại kỹ thuật chụp ảnh tiến trình khác nhau, dưới đây là các cách phân loại quan trọng nhất:

- Theo tiêu chí tính độc lập của trình chụp ảnh tiến trình có thể chia thành hai loại: Chụp ảnh tiến trình trong suốt (Transparent checkpointing) và Chụp ảnh tiến trình không trong suốt (Non-transparent checkpointing) [35]. Với loại Chụp ảnh tiến trình trong suốt ảnh chụp tiến trình được xử lý và lưu trữ bên ngoài không gian của trình ứng dụng được giám sát. Chương trình làm nhiệm vụ chụp ảnh tiến trình hoạt động độc lập với chương trình được giám sát. Loại Chụp ảnh tiến trình không trong suốt yêu cầu người viết trình chương trình ứng dụng phải phối hợp thực hiện trong mã chương trình ứng dụng, thông thường là qua việc gọi đến thư viện hỗ trợ việc chụp ảnh tiến trình và ảnh chụp được lưu trong không gian nhớ của chương trình ứng dụng. Như vậy Chụp ảnh tiến trình trong suốt giám sát không gian vùng nhớ của trình ứng dụng từ bên ngoài và ít làm ảnh hưởng đến trình ứng dụng hơn so với Chụp ảnh tiến trình không trong suốt.

- Theo tiêu chí thành phần của ảnh chụp, có 2 loại chính là Chụp ảnh tiến trình đầy đủ (Full checkpointing) và Chụp ảnh tiến trình gia tăng (Incremental checkpointing). Với Chụp ảnh tiến trình đầy đủ, toàn bộ không gian nhớ và các thông tin trạng thái khác của tiến trình sẽ được lưu trữ vào mỗi ảnh chụp tiến trình. Kỹ thuật chụp ảnh tiến trình gia tăng chỉ thực hiện lưu vào file ảnh tiến trình những thay đổi so với lần lưu trước đó nhằm mục đích tiết kiệm thời gian và tài nguyên của hệ thống. Phần lớn các nghiên cứu như [12],[37]-[40],[50]-[54] đều thực hiện cài đặt chụp ảnh tiến trình theo loại chụp ảnh tiến trình gia tăng.

- Theo tiêu chí về không gian thực hiện việc chụp ảnh, có 2 cấp độ là thực hiện ở không gian người sử dụng và ở không gian nhân. Một số chương trình và thư viện thực hiện chụp ảnh tiến trình nổi tiếng như: BLCR, DMTCP, CRIU sử dụng các mức độ khác nhau theo phân loại này.

BLCR (Berkeley Lab’s Linux Checkpoint/Restart) [41] sử dụng chụp ảnh ở không gian nhân để hỗ trợ cho việc dừng và khởi động lại chương trình MPI. BLCR được xây

dựng bằng cách bổ sung vào nhân Linux một mô đun hỗ trợ chụp ảnh tiến trình. Đồng thời BLCR cũng hỗ trợ cơ chế proxy (callbacks/proxy/hook) để tích hợp với các công cụ và thư viện khác như MPI.

DMTCP (Distributed MultiThreaded CheckPointing) [43][56] thuộc loại chụp ảnh tiến trình mức nhân. Tuy nhiên, thay vì thêm mô-đun mới vào nhân Linux. DMTCP sử dụng cơ chế *fork process* (tạo tiến trình con) và hàm `abort( )` – kết thúc ngay tiến trình - của hệ điều hành từ đó đón nhận ảnh chụp trạng thái tiến trình file core dump của hệ điều hành tạo ra cho chương trình bị kết thúc bất thường. Để chụp ảnh những ứng dụng mà trạng thái của chúng không có trong file core dump (ví dụ như file mô tả), DMTCP thực hiện kỹ thuật proxy (callbacks/proxy/hook) để bắt một số tín hiệu của hệ điều hành để theo dõi các thay đổi của ứng dụng bị chụp ảnh. Tương tự BLCR, DMTCP lưu tất cả các trạng thái của trình ứng dụng trong theo từng khoảng trong vài mili giây dẫn đến kích thước của file ảnh chụp rất lớn [43]. CRIU [44] là một dự án thực hiện xây dựng thư viện chụp ảnh và phục hồi trạng thái chương trình trên Linux ở không gian người sử dụng. Sử dụng công cụ này, chúng ta có thể đóng băng một ứng dụng đang chạy cũng như lưu ảnh tiến trình vào đĩa cứng và phục hồi lại trạng thái ứng dụng theo ảnh được lưu. CRIU sử dụng cơ chế `ptrace` [57][58][55] để kiểm soát và giám sát toàn bộ hoạt động của trình ứng dụng.

- Một cách phân loại khác, theo kỹ thuật lập trình có thể phân làm 3 loại chính: Page-based, hash-based và proxy-based được trình bày chi tiết trong mục 3.2.2.

3.2 Kỹ thuật chụp ảnh tiến trình gia tăng

Từ phiên bản 2, CAPE sử dụng Kỹ thuật chụp ảnh tiến trình gia tăng rời rạc (DICKPT) làm kỹ thuật cơ sở cho các thao tác cơ bản của nó. Kỹ thuật này là một cải tiến của Kỹ thuật chụp ảnh tiến trình gia tăng (ICKPT) và được phát triển thêm các tính năng để hỗ trợ cho CAPE. Việc chuyển từ CAPE đơn luồng sang CAPE đa luồng đòi hỏi nhiều thay đổi căn bản trong kỹ thuật này. Do đó, cần phải xem xét và nghiên cứu lại những nguyên lý cơ sở của tất cả các vấn đề liên quan.

3.2.1 Nguyên lý hoạt động của Kỹ thuật chụp ảnh tiến trình gia tăng

Ý tưởng đầu tiên của Kỹ thuật chụp ảnh tiến trình, được thực hiện bởi Kỹ thuật chụp ảnh tiến trình đầy đủ, khá đơn giản: 1) Mỗi ảnh chụp sẽ lưu lại toàn bộ dữ liệu trạng thái tiến trình, bao gồm không gian nhớ và các thông số trạng thái khác của tiến

trình; 2) Mỗi khi cần khôi phục lại trạng thái của tiến trình tương ứng với một thời điểm chụp ảnh nào chỉ cần nạp (inject) vào trạng thái tiến trình này những giá trị đã được lưu trữ trong ảnh chụp tương ứng. Như vậy, mỗi ảnh chụp sẽ tồn tại một cách độc lập. Đồng thời, dung lượng của mỗi ảnh chụp nói chung là lớn, từ vài chục MB đến cỡ GB. Nếu có nhiều ảnh chụp sẽ dẫn đến việc dung lượng lưu trữ là rất lớn. Tương tự cho trường hợp cần chuyển ảnh chụp tiến trình để cập nhật trạng thái cho một tiến trình khác qua mạng sẽ dẫn đến khối lượng dữ liệu truyền tải cũng sẽ rất lớn.

Để khắc phục nhược điểm trên, Kỹ thuật chụp ảnh tiến trình gia tăng sẽ không lưu toàn bộ dữ liệu trạng thái tiến trình vào mỗi ảnh chụp tiến nữa. Thay vào đó, mỗi ảnh chụp chỉ lưu những giá trị đã được thay đổi so với ảnh chụp ngay trước đó. Như vậy, ảnh chụp đầu tiên có thể là một ảnh chụp đầy đủ, nhưng các ảnh chụp sau đó chỉ là một phần giá trị trạng thái của tiến trình và thường là có kích thước nhỏ hơn đến rất nhỏ hơn kích thước của một ảnh chụp đầy đủ. Các ảnh chụp tiến trình được lưu trữ một cách độc lập. Khi muốn khôi phục lại trạng thái tiến trình tại một thời điểm nào đó, các ảnh tiến trình, kể từ ảnh đầu tiên đến ảnh được chụp tại thời điểm tương ứng với thời điểm đó được hợp (merge) với nhau, sau đó được nạp vào không gian nhớ của tiến trình.

3.2.2 Cơ chế phát hiện vùng nhớ bị thay đổi

Vấn đề mấu chốt nhất để thực hiện kỹ thuật chụp ảnh tiến trình gia tăng là làm sao phát hiện được vùng dữ liệu bị thay đổi của tiến trình đang chạy. Có nhiều cách tiếp cận để thực hiện việc phát hiện vùng nhớ dữ liệu của tiến trình. Phần lớn các nghiên cứu tiếp cận bằng kỹ thuật giám sát các trang nhớ (page-based techniques) của tiến trình như [12][37][38][39][41][42]. Kỹ thuật giám sát dữ liệu thay đổi trên các trang nhớ được hỗ trợ bởi phần cứng và hệ điều hành được thực hiện bằng cách khóa việc ghi (write-protect) vào trang nhớ, đồng thời mở khóa khi có hoạt động ghi lên trang nhớ đã bị khoá trước đó. Một số kỹ thuật khác cũng được sử dụng như: kỹ thuật sử dụng hàm băm vùng nhớ (hash-based techniques) để phát hiện vùng nhớ bị thay đổi; kỹ thuật bắc cầu (proxy-based techniques). Kỹ thuật hàm băm vùng nhớ [40] sử dụng một hàm băm để so sánh vùng nhớ nào đã bị thay đổi, từ đó thực hiện chụp ảnh tiến trình gia tăng. Kỹ thuật bắc cầu [43][44] sử dụng cơ chế giám sát proxy được hỗ trợ bởi hệ điều hành để kiểm soát chương trình truy cập đến vùng nhớ nào. Bản chất của kỹ thuật này thực hiện việc chặn sự kiện chương trình ứng dụng tương tác với hệ điều hành ở hàng đợi sự kiện

của hệ điều hành và thực hiện một số điều chỉnh nội dung sự kiện hoặc các thao tác khác nếu cần trước khi để sự kiện tiếp tục được hệ điều hành xử lý. Đây là cơ chế được hệ điều hành hỗ trợ, các chương trình gõ tiếng Việt cài trên hệ điều hành không tích hợp sẵn cơ chế gõ tiếng Việt được thực hiện theo cơ chế này.

Với kỹ thuật giám sát trang nhớ, các trang nhớ của chương trình ứng dụng được đặt vào trạng thái khoá cấm ghi (read-only/write-protect) trước thời điểm cần giám sát chụp ảnh tiến trình. Khi chương trình thực hiện thao tác ghi vào vùng nhớ này, hệ điều hành sẽ phát sinh ra một tín hiệu lỗi (đối với hệ điều hành Linux là tín hiệu SIGSEGV). Phần chương trình làm nhiệm vụ chụp ảnh tiến trình cần đón nhận tín hiệu này và thực hiện nhiệm vụ lưu lại địa chỉ và nội dung của trang nhớ phát sinh ra tín hiệu đồng thời chuyển trạng thái trang nhớ sang chế độ có thể ghi để trình ứng dụng tiếp tục chạy bình thường.

Đối với hệ điều hành Linux, việc khóa vùng nhớ có thể được thực hiện trong không gian nhân hoặc không gian người sử dụng. Chi tiết của các kỹ thuật này được trình bày trong các mục 3.2.3 và 3.2.4 bên dưới.

3.2.3 Kỹ thuật lập trình khoá/mở_khoá các trang nhớ ở mức nhân

Kỹ thuật này được dùng để phát triển các chương trình chụp ảnh tiến trình (checkpointer), thực hiện công việc chụp ảnh tiến trình và/hoặc khôi phục trạng thái tiến trình của một chương trình khác (được gọi là chương trình được theo dõi). Các chương trình này đòi hỏi phải có các hàm truy cập, cập nhật giá trị, thay đổi trạng thái,… vào các trang nhớ của các chương trình được theo dõi. Vì lý do an toàn, tránh sự can thiệp trực tiếp của một chương trình vào không gian nhớ của một chương trình khác, những hàm này không thể viết được ở không gian người sử dụng mà phải được viết ở mức nhân để có đủ quyền gọi những hàm khác của hệ điều hành cho phép can thiệp vào trang nhớ của một chương trình.

Về hình thức, các hàm trên có thể được phát triển theo 2 hướng: 1) viết các chức năng bổ sung trực tiếp vào nhân của hệ điều hành, việc này đòi hỏi phải biên dịch lại hệ điều hành, một công việc khá phức tạp và tốn rất nhiều thời gian; và 2) viết các chức năng ở dạng một trình điều khiển thiết bị (driver) và thêm vào hệ điều hành [41][42] , việc này không đòi hỏi phải biên dịch lại hệ điều hành, do đó đơn giản và tiết kiệm thời gian hơn cho người lập trình [38]. Ở cả 2 hướng, sau khi đã có các hàm cơ sở cần thiết

ở mức nhân, chương trình chụp ảnh tiến trình chính sẽ được viết ở mức không gian người dùng và gọi các hàm cơ sở ở mức nhân để xử lý các truy cập và can thiệp vào không gian nhớ của chương trình được theo dõi. Nghiên cứu của chúng tôi tập trung phát triển theo hướng thứ 2, và được trình bày chi tiết hơn ở bên dưới.

Để có thể hình dung rõ hơn về cách thức một chương trình có thể can thiệp vào không gian nhớ của một chương trình khác ở mức nhân, ta xét một ví dụ cụ thể về một hàm của trình chụp ảnh tiến trình thực hiện thiết lập trạng thái có thể ghi cho một trang nhớ chứa một địa chỉ xác định của một chương trình được theo dõi. Ví dụ này được viết cho trường hợp bộ nhớ ảo được phân chia 3 cấp.

Để đặt cờ Read/Write (Đọc/Ghi) của một trang nhớ chứa địa chỉ `addr` trong không gian nhớ của tiến trình ứng dụng (tương ứng với chương trình được theo dõi) có chỉ số là `pid`, các bước chính sẽ là:

Tìm đối tượng bộ nhớ tiến trình (`mm_struct object`*).* Đối tượng này là manh mối khởi đầu cho việc truy cập hầu hết các đối tượng quản lý bộ nhớ tiến trình. Từ chỉ số của tiến trình, có thể truy cập đối tượng `task_struct` tương ứng và trường `mm` của đối tượng này, trường này trỏ tới đối tượng bộ nhớ cần tác động.

Kiểm tra tính hợp lệ của địa chỉ. Về mặt lý thuyết, không gian bộ nhớ ảo của một tiến trình có 4GB (địa chỉ 32 bít), có địa chỉ tổng thể từ `0x00000000` đến `0xFFFFFFFF` và từ địa chỉ `start_code` đến `0xC0000000` dành cho không gian người dùng. Tuy nhiên, nhân của hệ điều hành chỉ truy cập các vùng nhớ địa chỉ ảo (`vma`) đối với các vùng đã thật sự được hệ điều hành cấp phát cho tiến trình. Do đó, việc kiểm tra tính hợp lệ của một địa chỉ sẽ đồng nghĩa với việc kiểm tra địa chỉ này có nằm trong một `vma` nào đó hay không.

Tìm trang nhớ tương ứng và đặt trạng thái trang nhớ về trạng thái có thể ghi. Đối với bộ nhớ ảo được quản lý 3 cấp, lần lượt tìm các đối tượng tương ứng trong các bảng chỉ mục Page Global Directory, Page Middle Directory và Page Table. Có 3 lưu ý quan trọng trong các bước này. Thứ nhất là việc khoá/mở_khoá bảng Page Table (Bảng quản lý trang) mỗi khi có thay đổi trên thành phần của nó. Điều này nhằm tránh các xung đột với các truy cập khác trên các hệ thống đa bộ xử lý và/hoặc bộ xử lý đa lõi. Điều thứ hai liên quan đến việc gọi hàm `handle_mm_fault()` của nhân để xử lý trường hợp vùng nhớ là hợp lệ nhưng lại không nằm trong bộ nhớ chính (RAM). Lưu ý cuối cùng

liên quan đến việc cập nhật bảng Page Table từ vùng đệm, sau khi được chỉnh sửa, để đảm bảo là cập nhật này được khả dụng ở các thao tác về sau. Hàm `pte_mkwrite()` được sử dụng để đặt trang nhớ về trạng thái có thể ghi.

Dưới đây là trích đoạn mã của hàm thực hiện thiết lập trạng thái có thể ghi cho một trang nhớ trong chương trình chụp ảnh tiến trình DICKPT. Hàm này nhận qua tham số chỉ số của tiến trình và một địa chỉ, sau đó thực hiện đặt trang nhớ chứa địa chỉ này về trạng thái có thể ghi.

```
int clear_write_protect(unsigned int pid, unsigned long addr){
    struct task_struct *t, *task = NULL;
    for_each_process(t) //find the task associated with process
        if (t->pid == pid) task = t; //search the task object
    if(task == NULL){ printk("pid invalid"); return 1; }
    mm = task->mm; //the memory object
    vma = find_vma(mm, addr); //the virtual memory area
    if(!vma){printk("addr invalid");
       return 2;}//this address is not allocated
    spin_lock(&mm->page_table_lock); //lock the Page Table
    pgd = pgd_offset(mm, addr); //search the PGD
    if (pgd_none(*pgd)) { ret = 3; goto out; }
    pmd = pmd_offset((pud_t *)pgd, addr); //search the PMD
    if (pmd_none(*pmd)){ //In case of setting a new value for the brk
                         //(end of heap), the PMD may not be exist.
                         //So, let the kernel try to process first.
        spin_unlock(&mm->page_table_lock); //unlock the PT before
                                    // letting the kernel to process
        ret = my_handle_mm_fault(mm, vma, addr, 0); //this will call
                           //the handle_mm_fault( ) kernel function.
        spin_lock(&mm->page_table_lock);
    }
    if(pmd_none(*pmd)) { ret = 4; goto out; }
    pte = pte_offset_map(pmd, addr); //search the PTE
    if (pte_present(*pte)){ //page is in the main memory
        *pte = pte_mkwrite(*pte); //set its status to writable
        flush_tlb_page(vma, addr); //flush the cache to update the PT
    }else{//page isn't in main memory, let the kernel to process first
        vma->vm_flags |= VM_WRITE;
        pte_unmap(pte);
        spin_unlock(&mm->page_table_lock);
        ret = my_handle_mm_fault(mm, vma, addr, 0);
        spin_lock(&mm->page_table_lock);
        vma->vm_flags &= _VM_WRITE;
        if(ret == 0){
            pte = pte_offset_map(pmd, addr);
            *pte = pte_mkwrite(*pte);
            flush_tlb_page(vma, addr);
        }else{
            pte_unmap(pte);
            goto out;
        }
    }
    pte_unmap(pte);
    spin_unlock(&mm->page_table_lock);
    return ret;
    out:
        printk("error: %d\n", ret);
        spin_unlock(&mm->page_table_lock);
```

```
        return ret;
}
```

Đối với các nhân sử dụng phân trang 5 mức cần có thêm 2 bước duyệt trung gian giữa bước tìm `pgd` và `pmd` để thực hiện việc tìm đến 1 trang nhớ tương ứng với `pte` trong đoạn mã trên.

3.2.4 Kỹ thuật lập trình khóa/mở khóa trong không gian người sử dụng

Như đã đề cập ở phần trên, ở mức không gian người sử dụng, một tiến trình chỉ có thể truy cập trực tiếp và chỉnh sửa vào không gian nhớ của chính nó. Để thực hiện việc khoá/mở_khoá một vùng nhớ, có thể sử dụng các hàm tương ứng của hệ điều hành hoặc của chính ngôn ngữ lập trình bậc cao (nếu có). Hàm phổ biến được sử dụng là `mprotect` [59]. Cú pháp gọi hàm như sau:

```
int mprotect(void *addr, size_t len, int prot)
```

Trong đó: `addr`: địa chị vùng nhớ bắt đầu khóa/mở khóa, `len`: kích thước vùng nhớ cần khóa/mở khóa bắt đầu tính từ addr, `prot` là cờ truy cập.

Lệnh thực hiện khóa:

```
mprotect(void *addr, size_t len, PROT_READ)
```

Và lệnh thực hiện mở khóa:

```
mprotect(addr, SIZE, PROT_WRITE | PROT_EXEC | PROT_READ)
```

Để bắt được tín hiệu lỗi xảy ra khi quá trình ghi lên vùng nhớ bị khóa chương trình cần đăng ký kiểm soát tín hiệu `SIGSEGV` của hệ thống như các dòng lệnh trong Hình 3.8. Khi có tín hiệu `SIGSEGV` xảy ra trình chụp ảnh tiến trình thực hiện mở khóa vùng nhớ để chương trình ứng dụng thực hiện tiếp và thực hiện việc thu thập dữ liệu làm ảnh chụp tiến trình. Tất cả những việc xử lý khi tín hiệu `SIGSEGV` sinh ra cần được đặt trong hàm `Sigsegv_Handler`.

```
sa.sa_flags = SA_SIGINFO;
sigemptyset(&sa.sa_mask);
sa.sa_sigaction = Sigsegv_Handler;
if (sigaction(SIGSEGV, &sa, NULL) == -1)
            handle_error("sigaction");
```

Hình 3.8. Trích đoạn mã đăng ký nhận và xử lý tín hiệu SIGSEGV

3.2.5 So sánh kỹ thuật khóa/mở khóa ở không gian nhân và không gian người sử dụng

Căn cứ vào nguyên lý được trình bày ở những phần trên, sự khác biệt của 2 kỹ thuật được trình bày trong Bảng 3.1.

Bảng 3.1. Bảng so sánh định tính giữa hai mức khóa vùng nhớ

Tiêu chí	**Khóa ở không gian nhân**	**Khóa ở không gian người sử dụng**
Không gian khóa	Có thể khóa nhiều chương trình khác	Khóa vùng nhớ của chương trình chính mình
Khả năng lỗi khi chuyển qua kernel mới	Rất cao	Thấp
Tùy biến khóa	Cao	Thấp
Việc xây dựng chương trình	Phức tạp	Đơn giản
Tính trong suốt (transparent) với trình ứng dụng	Cao	Thấp
Ảnh hưởng đến vùng nhớ của trình ứng dụng	Ít	Nhiều
Tốc độ *	Cao	Cao

* Chỉ tiết ở được đánh giá ở phần sau

Bảng 3.1 trình bày so sánh đánh giá hai kỹ thuật khoá/mở_khoá trên một số tiêu chí quan trọng nhất. Mỗi kỹ thuật có ưu nhược điểm riêng. Do vậy, cần dựa vào nhu cầu và đặc điểm cụ thể của công việc để lựa chọn kỹ thuật phù hợp.

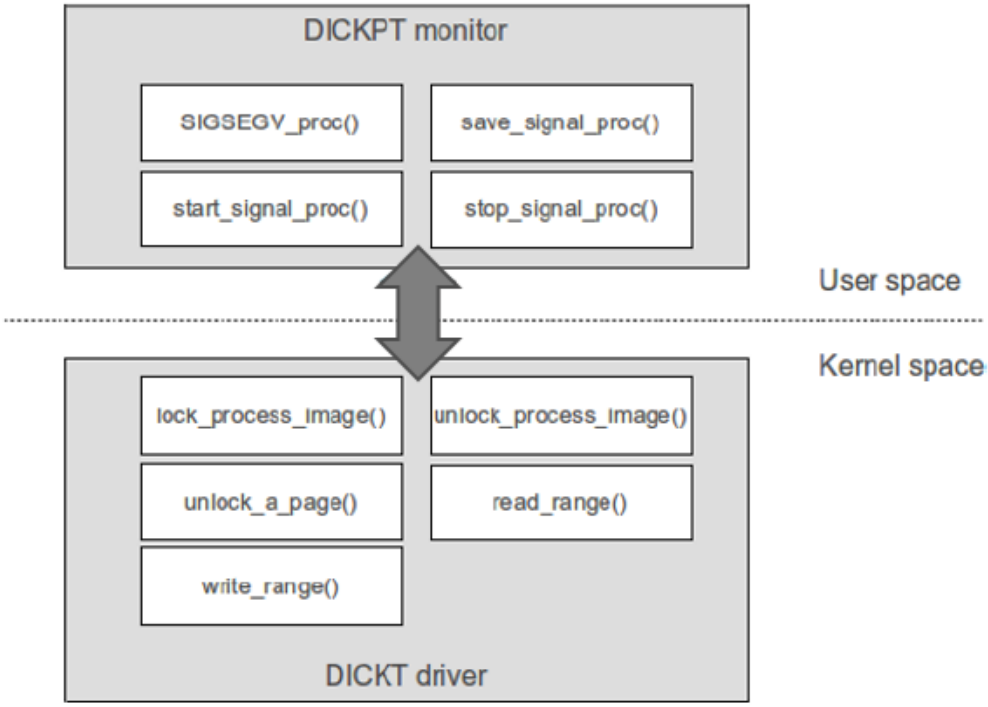
3.3 Kỹ thuật chụp ảnh tiến trình gia tăng rời rạc cho CAPE đa luồng

3.3.1 Kỹ thuật chụp ảnh tiến trình gia tăng rời rạc

CAPE sử dụng kỹ thuật chụp ảnh tiến trình gia tăng rời rạc. Nguyên tắc hoạt động của kỹ thuật này đã được trình bày trong phần giới thiệu về CAPE đơn luồng ở trang 34.

Trong phần này, chúng ta tìm hiểu thêm một khía cạnh khác là cấu trúc của các chương trình DICKPT đã được phát triển trước đây, như được trình bày như trong Hình 3.9. Theo cấu trúc này, chương trình chụp ảnh tiến trình được cấu thành từ 2 thành phần

cơ bản: 1) DICKPT driver là một driver bổ sung vào nhân của hệ điều hành, hoạt động trong không gian nhân, chủ yếu chứa các hàm thực hiện việc khoá/mở_khoá toàn bộ hoặc 1 trang nhớ của tiến trình, đọc/ghi dữ liệu lên 1 vùng địa chỉ trong không gian nhớ của tiến trình; và 2) DICKPT monitor (trình giám sát) là một chương trình hoạt động trong không gian nhớ người sử dụng. Chương trình này sử dụng thực hiện việc giám sát tiến trình được theo dõi, xử lý vào những yêu cầu dưới dạng các tín hiệu hệ thống (system call) được gửi đến và gọi các hàm tương ứng của DICKPT driver để thực hiện.



Hình 3.9. Sơ đồ tổ chức chương trình DICKPT trong CAPE đơn luồng

3.3.2 Chọn lựa không gian phát triển trình chụp ảnh tiến trình

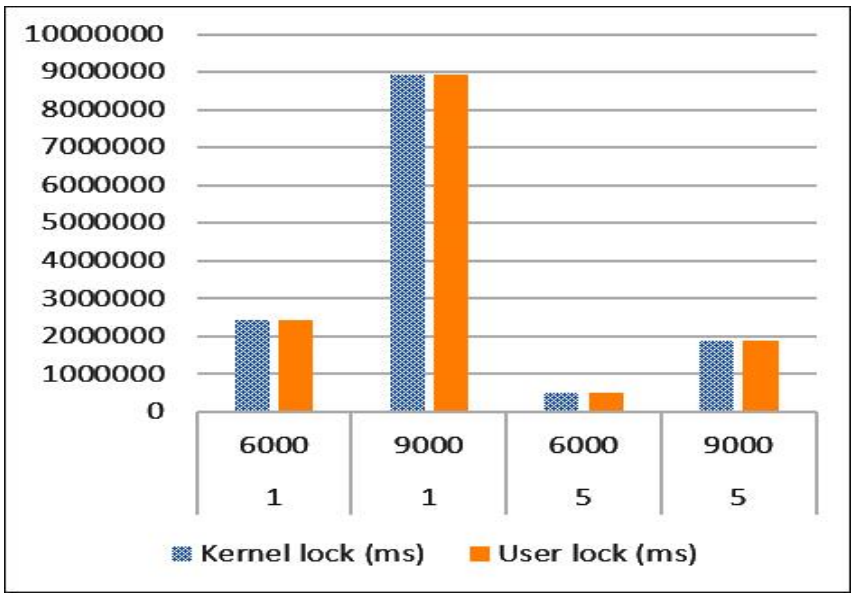
Như đã trình bày ở phần dẫn nhập của chương, khi chuyển qua mô hình hoạt động mới của CAPE đa luồng, công cụ chụp ảnh tiến trình gia tăng rời rạc đã được phát triển trước đó cho CAPE đơn luồng không còn sử dụng được nữa. Chúng tôi đã tiến hành sử dụng thử công cụ này trên mô hình mới, ngay lập tức các chương trình liên quan bị treo. Theo nhận định ban đầu, tình trạng này là do xung đột giữa các sự kiện khi các luồng song song trong tiến trình cùng truy cập vào các trang nhớ bị khoá, cũng như việc khoá và truy cập các trang nhớ vùng stack, trước đây vốn chỉ là 1 vùng độc lập cho toàn bộ tiến trình, nay lại bao gồm nhiều vùng khác nhau cho những luồng song song. Hơn nữa, phiên bản DICKPT cho CAPE đơn luồng được xây dựng trên phiên bản hệ điều hành đã cũ, không còn được hỗ trợ, nên khi chuyển sang phiên bản mới cũng cần có rất nhiều

thay đổi phức tạp. Đặc biệt, nếu muốn giữ nguyên cấu trúc hoạt động của trình chụp ảnh tiến trình hoạt động ở mức nhân thì việc thay đổi lại càng khó khăn. Do đó, chúng tôi đã xem xét và thử nghiệm cả phương án phương án phát triển trình chụp ảnh tiến trình ở mức không gian người sử dụng để hạn chế bớt độ phức tạp khi xây dựng, cũng như hạn chế bớt sự đòi hỏi phải cập nhật khi nâng cấp hệ điều hành. Tuy nhiên, trước khi chuyển hướng, một câu hỏi được đặt ra là liệu trình chụp ảnh tiến trình được xây dựng ở mức không gian người dùng có làm giảm hiệu năng của hệ thống hay không. Để trả lời câu hỏi đó, chúng tôi đã thử nghiệm cài đặt cả 2 hướng để so sánh tốc độ của chúng khi sử dụng làm công cụ cơ sở của CAPE đơn luồng.

Thử nghiệm được tiến hành với bài toán nhân 2 ma trận vuông, chạy trên hệ điều hành Ubuntu 18.04 – Kernel 5.0.0.29.x, hệ thống các máy tính được kết nối qua mạng LAN 100 Mbps[4], các máy tính có cấu hình CPU Intel(R) Core(TM) i3-2100 @3.1GHz RAM 8GB, SSD 240 GB. Có 2 cấu hình hệ thống là sử dụng là sử dụng 5 máy tính (1 nút chính và 4 nút phụ) và sử dụng 1 máy tính (chỉ có 1 nútchính). Cấu hình thứ nhất là để thử nghiệm trên hệ thống chạy CAPE thông thường, với cấu trúc master-slave. Cấu hình thứ 2, khá đặc biệt, chỉ có mỗi nút chính (chỉ chạy trên một máy) nhưng vẫn áp dụng cơ chế chụp ảnh tiến trình để lấy kết quả chương trình CAPE. Mục đích của cấu hình này là để loại trừ yếu tố ảnh hưởng của thời gian phân chia công việc và truyền nhận dữ liệu trên mạng, điều có thể ảnh hưởng đến hiệu năng của hệ thống.

Kết quả thực nghiệm được thể hiện ở Hình 3.10.

[4] Các máy tính cùng lớp mạng được nối chung cùng 1 switch 48 ports 10/100 Mbps, dây mạng CAT5.



Hình 3.10. So sánh hiệu năng của hai kỹ thuật khóa vùng nhớ khi áp dụng cho CAPE

Trên Hình 3.10, trục tung chỉ thời gian chạy theo đơn vị milliseconds; trên trục hoành, ở hàng thứ nhất (6000/9000) biểu thị kích thước ma trận vuông và hàng thứ 2 hai biểu thị số máy tính (1/5) của hệ thống thử nghiệm. Các kết quả cho thấy một cách rõ ràng là hai phương pháp khóa/mở_khóa đều cho hiệu suất về thời gian tương đương nhau.

Để giải thích cho kết quả này chúng tôi đã xem lại mã nguồn của hàm `mprotect` [59], được sử dụng trong trình chụp ảnh tiến trình mức không gian người sử dụng và nhận thấy rằng để khóa/mở_khóa một trang nhớ hàm này cũng cần tuân thủ nguyên tắc của hệ điều hành và thực hiện đoạn mã duyệt tuần tự tất các mức phân trang của vùng nhớ để thực hiện việc khóa/mở_khóa bao gồm trích đoạn mã như đã trình bày ở mục 3.2.3. Do đó, về thực chất, cả hai kỹ thuật này đều chạy những đoạn mã có độ phức tạp tương đương, dẫn đến hiệu suất của chúng là tương đương nhau. Tuy nhiên, do hàm `mprotect()` được cung cấp bởi hệ thống, nên phần mã của hàm này cũng tự động được cập nhật trên mỗi phiên bản của hệ điều hành, do đó người phát triển trình chụp ảnh tiến trình không cần cập nhật theo.

Tuy nhiên, có một khía cạnh khác cũng cần xem xét, liên quan đến một điểm khác nhau quan trọng giữa hai kỹ thuật này như đã trình bày trong mục 3.2.3. là kỹ thuật chụp ảnh tiến trình thực hiện trong không gian nhân có thể hoạt động trong suốt (transparent) với trình ứng dụng bị giám sát. Kết quả này có được nhờ vào yếu tố ảnh chụp và các đoạn mã thực hiện công việc chụp ảnh được thực hiện bên ngoài không

gian nhớ của trình ứng dụng được giám sát. Ngược lại, với kỹ thuật khóa vùng nhớ ở ở không gian người sử dụng, quá trình giám sát và xử lý ảnh chụp tiến trình thực hiện chính trong không gian nhớ của trình ứng dụng. Về mặt nguyên lý, vấn đề này có thể làm ảnh hưởng đến trình ứng dụng, vì không gian nhớ của chương trình khi hoạt động không giống như trong trường hợp nguyên thủy nữa, hay nói cách khác, hoạt động của chương trình có thể khác với chương trình nguyên thủy trong một số trường hợp đặc biệt nào đó. Tuy nhiên, ở mức độ được sử dụng để thử nghiệm CAPE ở mức độ căn bản, điều này là chấp nhận được. Do đó, chúng tôi đã chọn phương án sử dụng không gian nhớ người sử dụng để phát triển trình chụp ảnh tiến trình cho CAPE đa luồng.

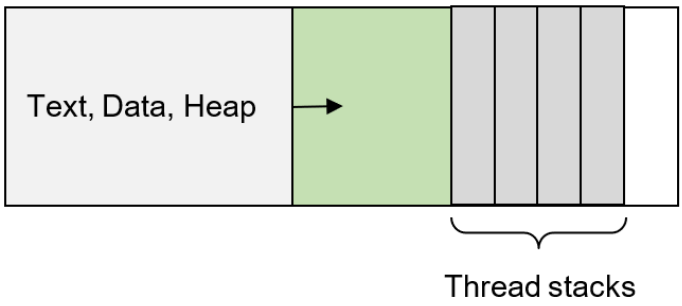
3.3.3 Các thách thức khi chuyển từ trình chụp ảnh tiến trình đơn luồng sang chụp ảnh tiến trình đa luồng

Thách thức thứ nhất: Sự khác biệt địa chỉ giữa các luồng khi đồng bộ ảnh chụp tiến trình

Nguyên lý hoạt động của CAPE dựa trên kỹ thuật chụp tiến trình. Nguyên lý này chỉ thực hiện được khi không gian nhớ của các tiến trình của chương trình ứng dụng trên các nút là đồng bộ với nhau. Nghĩa là một biến V1 của chương trình P1 được chạy ở nút chính được lưu tại địa chỉ M1, khi chương trình P1 được chạy ở các nút phụ cũng phải được lưu tại địa chỉ M1. Điều này luôn được đảm bảo ở CAPE đơn luồng, do hệ điều hành tổ chức vùng nhớ cho trình ứng dụng theo cơ chế bộ nhớ ảo có địa chỉ liên tục và bắt đầu từ địa chỉ 0 [46][47]. Do vậy một chương trình chạy trên một hệ điều hành như nhau, trên nhiều máy tính khác nhau đều có không gian nhớ giống nhau.

Đối với hệ điều hành Linux 32 bit, mỗi tiến trình luôn được cấp phát 4 GB bộ nhớ ảo khi chạy, trong đó gồm 1GB cho không gian nhân và 3 GB cho không gian người sử dụng. Không gian vùng nhớ ảo này được tổ chức thành các trang nhớ liên tục, mỗi trang nhớ có kích thước bằng nhau (4KB). Hơn nữa, tại mỗi điểm trên trục thời gian thực hiện của chương trình, khi chạy trên cùng một phiên bản của hệ điều hành các dữ liệu trong các trang nhớ này là hoàn toàn giống nhau. Tuy nhiên, điều này không còn đúng hoàn toàn khi chạy một chương trình ở dạng đơn luồng và khi chạy đa luồng. Khi một luồng được tiến trình tạo ra, một stack dành riêng cho tiến trình được cấp phát trong vùng stack của tiến trình, như được minh họa như trong Hình 3.11. Kích thước của vùng

nhớ này có thể khác nhau ở các hệ điều hành khác nhau. Ở hệ điều hành Linux x86-32, mặc định mỗi thread được cấp phát 2MB [74].



Hình 3.11. Minh họa việc cấp phát vùng nhớ cho tiến trình đa luồng

Quay lại với mô hình hoạt động của CAPE đa luồng, các tiến trình chạy ở các nút ở dạng đa luồng. Điều này dẫn đến hệ quả là không gian nhớ của tiến trình ứng dụng không còn giống nhau trên các nút nữa, dẫn đến nguyên tắc địa chỉ giống không còn đúng để thực hiện đồng bộ ảnh chụp tiến trình. Đối với thách thức này, chúng tôi đã xem xét hai giải pháp thực hiện.

Giải pháp đầu tiên là thực hiện ảnh chụp tiến trình qua 2 mức. Mức 1 giám sát và chụp ảnh tiến trình chính, mức 2 giám sát và chụp ảnh các luồng. Do các luồng không cùng địa chỉ nên muốn thực hiện ảnh chụp ở mức 2 đồng bộ được với nhau cần phải ánh xạ lại địa chỉ, việc này khá phức tạp và có thể gây giảm hiệu năng của chương trình.

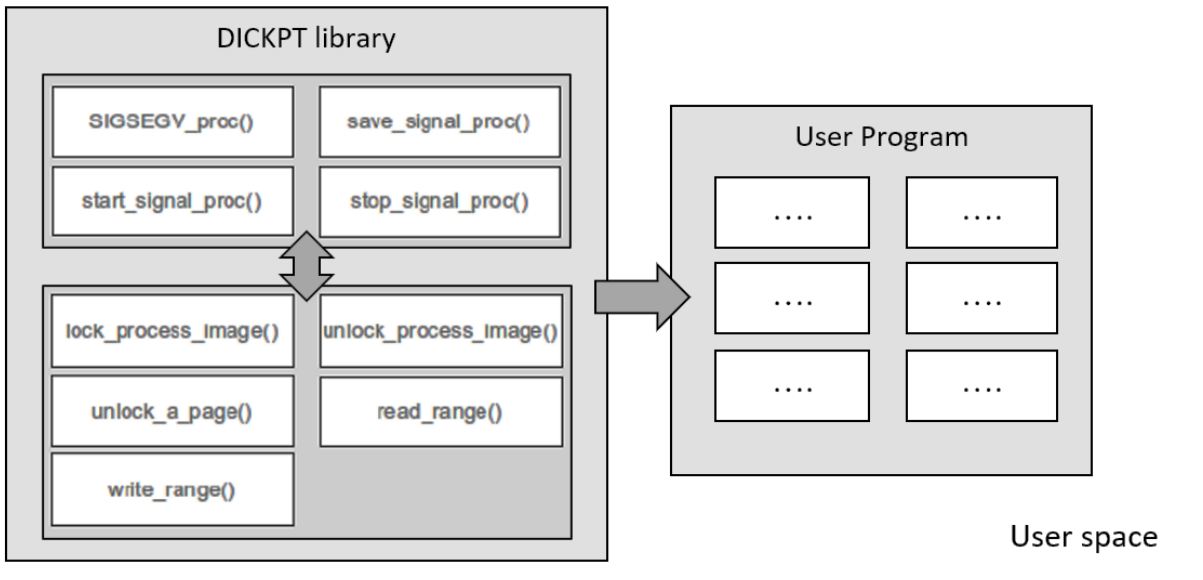
Giải pháp thứ hai, được thực hiện dựa trên logic hoạt động của OpenMP ở đoạn mã song song, tiến trình chính (chỉ có 1 luồng duy nhất – luồng ban đầu) tạo nhiều luồng phụ để thực hiện công việc, sau đó kết quả được cập nhật và chỉ được giữ lại trong các vùng nhớ của luồng ban đầu này. Do đó, để trích rút kết quả cuối cùng ở các nút phụ chỉ cần thực hiện việc chụp ảnh của luồng ban đầu, khi các luồng phụ đã thực hiện xong việc tính toán. Khi đó, việc chụp ảnh thực chất chỉ chụp vùng dữ liệu của tiến trình và vùng stack của luồng ban đầu. Tuy nhiên, cũng cần phải giải quyết một vấn đề khác là trình chụp ảnh tiến trình ở các nút phụ cần loại bỏ việc giám sát vùng stack của các luồng phụ để không dẫn đến xung đột phát sinh và các vấn đề về khác biệt địa chỉ.

Thách thức thứ hai: giới hạn kỹ thuật của hệ điều hành trong việc một tiến trình thực hiện giám sát một tiến trình tiến trình khác

Phiên bản DICKPT hiện nay sử dụng cơ chế giám sát độc lập. Nghĩa là trình giám sát là một chương trình riêng độc lập với trình ứng dụng, sử dụng một driver hoạt động ở mức nhân để khóa vùng nhớ dữ liệu của ứng dụng và sử dụng hàm `ptrace` [57][58] của hệ điều hành để giám sát trình ứng dụng và thực hiện chụp ảnh tiến trình. Tuy nhiên cơ chế này không thể giám sát được các tiến trình có nhiều luồng, thể hiện ở việc hệ thống tại các nút phụ bị treo ngay khi chuyển đổi các chương trình ứng dụng từ dạng đơn luồng sang đa luồng. Khắc phục vấn đề này là một việc rất phức tạp và đòi hỏi sự thay đổi thường mỗi lần nâng cấp hệ điều hành. Do đó, chúng tôi đã thay đổi việc phát triển trình chụp ảnh tiến trình của CAPE từ mức không gian nhân sang mức không gian người sử dụng, với các thay đổi về cấu trúc và đặc tính như đã trình bày ở mục trên.

3.4 Kết quả phát triển Trình chụp ảnh tiến trình mới cho CAPE đa luồng

Với những nghiên cứu, phân tích, thử nghiệm vừa được nêu trên, trình chụp ảnh tiến trình mới cho CAPE đa luồng được xây dựng dưới dạng một thư viện cung cấp các hàm có chức năng tương như các hàm của DICKPT monitor và DICKPT driver trong trình chụp ảnh tiến trình của CAPE đơn luồng trước đây. Cấu trúc này được minh họa trong Hình 3.12.

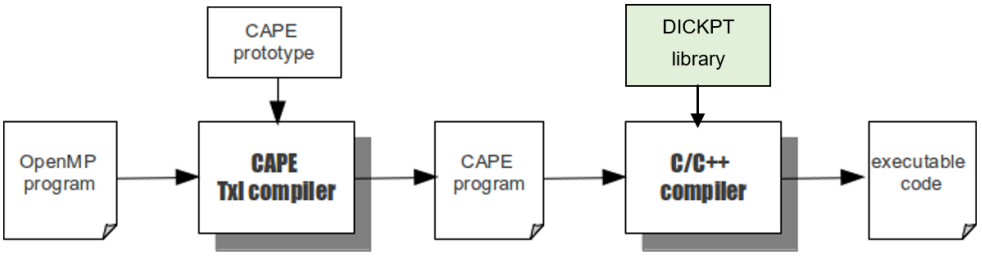


Hình 3.12. Kiến trúc trình chụp ảnh tiến trình DICKPT cho CAPE đa luồng

Do đã gộp cả phần monitor và driver vào trong cùng 1 thư viện DICKPT trong không gian người sử dụng, nên hoạt động của các hàm cũng đã đơn giản đi khá nhiều. Việc xử lý các tín hiệu hệ thống được thực hiện ngay trong bản thân chương trình bằng cách đơn giản là gọi các hàm xử lý tương ứng, ở ngay trong cùng một thư viện và tất

cả đều ở mức không gian người sử dụng. Chẳng hạn như hàm `SIGSEVG_proc()` đơn giản chỉ là gọi hàm `unlock_a_page()` trong cùng DICKPT library.

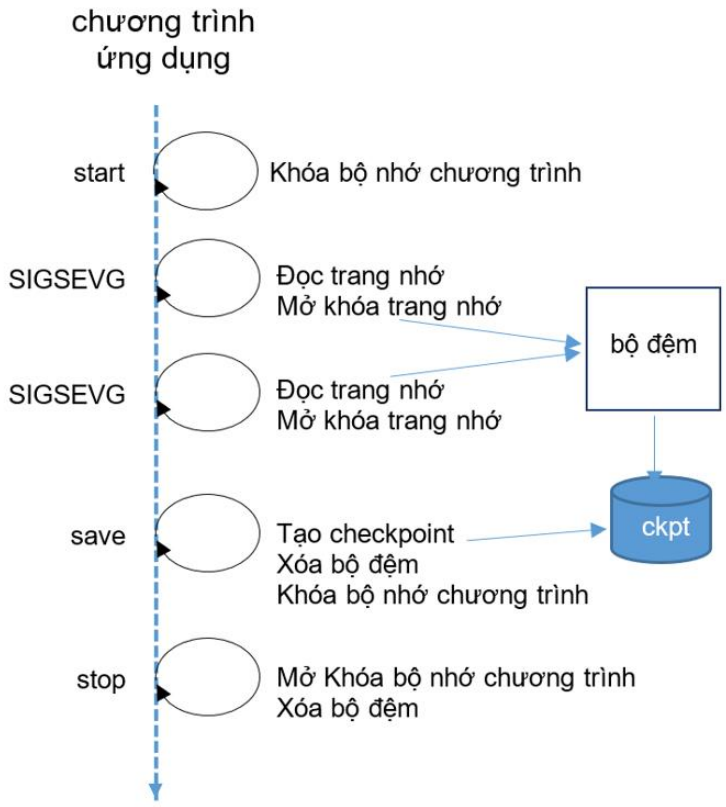
Thư viện này được trình biên dịch C/C++ sử dụng khi biên dịch chương trình nguồn dạng CAPE thành chương trình dạng mã máy. Do được phát triển ở dạng một thư viện liên kết tĩnh nên sau khi trình ứng dụng đã được biên dịch xong có thể được chạy trên nền tảng CAPE mà không cần có sẵn thư viện CAPE này nữa. Quy trình biên dịch của chương trình được biểu diễn trên Hình 3.13. Quy trình này có điểm khác biệt nhỏ, so với quy trình biên dịch chương trình CAPE đơn luồng trước đây (Hình 1.7) ở bước sử dụng bộ biên dịch C/C++, lúc này cần phải có thư viện DICKPT.



Hình 3.13. Quy trình biên dịch chương trình OpenMP thành chương trình CAPE đa luồng

Với việc chuyển hướng xây dựng trình ứng dụng sang ở mức không gian người sử dụng với cấu trúc như được nêu ở Hình 3.14, nguyên tắc hoạt động của trình chụp ảnh tiến trình trước đây (Hình 1.12) cũng có sự thay đổi nhỏ, trong đó hoạt động của trình ứng dụng và trình chụp ảnh tiến trình được thực hiện trong một tiến trình duy nhất. Nguyên tắc mới này được trình bày trong Hình 3.14. Tuy nhiên, sự thay đổi này không làm thay đổi nguyên lý hoạt động của CAPE, như đã được trình bày trong Hình 2.7.

Có thể thấy, trình chụp ảnh tiến trình mới có cấu trúc và hoạt động đơn giản hơn so với trình chụp ảnh tiến trình trước đây. Tuy nhiên, như đã được so sánh trong mục 3.2.5, nhược điểm của trình chụp ảnh tiến trình mới là tính không trong suốt đối với chương trình ứng dụng, thể hiện ở chỗ các hàm của thư viện DICKPT được thêm vào mã chương trình ứng dụng và những dữ liệu của quá trình chụp ảnh tiến trình cũng được xử lý trong không gian nhớ của tiến trình ứng dụng.



Hình 3.14. Nguyên tắc hoạt động của trình chụp ảnh tiến trình trong CAPE đa luồng

Trình chụp ảnh tiến trình mới đã được sử dụng để thử nghiệm CAPE với mô hình hoạt động mới – CAPE đa luồng – với các kết quả được trình bày ở Chương 2.

3.5 Tiểu kết chương III

Chương này trình bày những kiến thức liên quan để xây dựng phiên bản mới cho Kỹ thuật chụp ảnh tiến trình gia tăng rời rạc, để có thể chụp ảnh được các tiến trình đa luồng, phục vụ cho hoạt động trong phiên bản CAPE đa luồng mới.

Công việc liên quan đến việc quản lý bộ nhớ, chụp ảnh tiến trình là một trong những vấn đề phức tạp nhất của lập trình hệ thống. Luận án đã phát triển công cụ chụp ảnh tiến trình phù hợp với CAPE đa luồng ở không gian người sử dụng với ưu điểm là tính ổn định cao với các phiên bản khác nhau của OS do chỉ gọi lại các hàm thư viện phổ thông của OS cung cấp và các thử nghiệm ban đầu đã cho kết quả tốt.

Các kết quả của chương này được công bố ở công trình [CT4], [CT5].

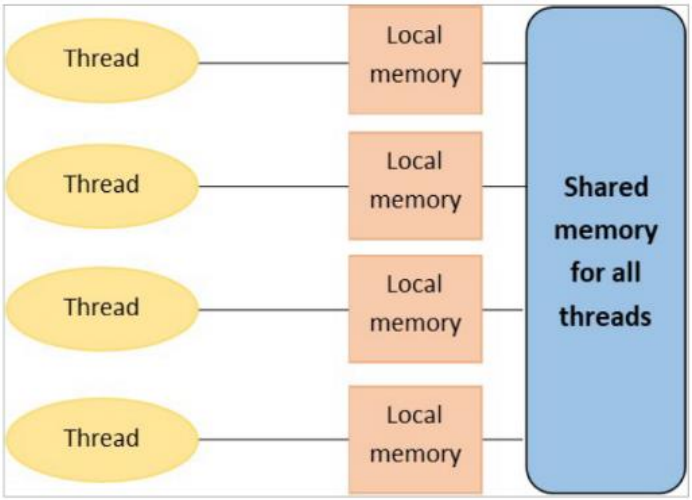
CHƯƠNG 4 XỬ LÝ CÁC VẤN ĐỀ CHIA SẺ DỮ LIỆU CỦA CAPE ĐA LUỒNG

CAPE là một hướng tiếp cận để cài đặt OpenMP - giao diện lập trình chuẩn cho lập trình song song trên các hệ thống sử dụng bộ nhớ chia sẻ - lên các hệ thống sử dụng bộ nhớ phân tán. Để CAPE có thể chuyển đổi hoàn chỉnh OpenMP trên hệ thống bộ nhớ phân tán, CAPE cần phải xử lý các vấn đề về dữ liệu chia sẻ của OpenMP và thực nghiệm được tính đúng của việc xử lý đó. Trong mục 1.7.5 đã trình bày CAPE sử dụng cấu trúc dữ liệu SSD {(addr, size, values)} để lưu trữ dữ liệu ảnh chụp tiến trình. Khi các nút phụ gửi ảnh chụp tiến trình về nút chính trong nội dung dữ liệu địa chỉ (addr) trùng nhau sẽ xử lý như thế nào ưu tiên nút phụ nào trước, cũng như các vấn đề liên quan khác. Chương này trình bày cách giải quyết vấn đề này một cách chi tiết.

4.1 Sự khác nhau của mô hình chia sẻ dữ liệu của OpenMP trên hệ thống bộ nhớ chia sẻ và CAPE trên hệ thống bộ nhớ phân tán

4.1.1 Chia sẻ dữ liệu của OpenMP

Một cách tổng quan, OpenMP sử dụng mô hình thực hiện đa luồng để vận hành các cấu trúc song song OpenMP, trong đó các luồng cùng chia sẻ không gian nhớ chung của chương trình. Tuy nhiên, để tăng tốc độ xử lý, OpenMP áp dụng mô hình chia sẻ dữ liệu đồng bộ trễ (Relaxed-Consistency - RC) [2]. Với mô hình này, các luồng có thể sử dụng bộ nhớ cục bộ để tăng hiệu suất truy xuất bộ nhớ. Việc đồng bộ không gian bộ nhớ của các luồng được thực hiện tự động tại lúc bắt đầu và kết thúc của mỗi cấu trúc vòng lặp song song hoặc có thể được chỉ định cụ thể qua các chỉ thị `flush`. Hình 4.1 minh họa mô hình tổ chức vùng nhớ cục bộ và chia sẻ của OpenMP.



Hình 4.1. Mô hình vùng nhớ chia sẻ và cục bộ trong OpenMP

Ngữ cảnh sẽ thay đổi hoàn toàn khi sử dụng OpenMP trên mô hình đa tiến trình, vận hành trên các hệ thống bộ nhớ phân tán, trong đó mỗi tiến trình sử dụng một không gian nhớ riêng, trên các máy tính khác nhau. Về cơ bản, cách thức này là tương thích với mô hình bộ nhớ RC của OpenMP, với yêu cầu kèm theo là phải xử lý được được các chỉ thị liên quan đến dữ liệu chia sẻ. Cần nhấn mạnh thêm là chính nhờ mô hình RC của OpenMP - mô hình chia sẻ dữ liệu đồng bộ trễ có sử dụng bộ không gian bộ nhớ của các luồng riêng nên CAPE có thể chuyển đổi được OpenMP sang hệ thống bộ nhớ phân tán.

4.1.2 Chia sẻ dữ liệu của CAPE

OpenMP sử dụng mô hình bộ nhớ chia sẻ đồng bộ trễ, trong đó tất cả các luồng OpenMP đều có quyền truy cập vào một nơi để lưu trữ và truy xuất các biến, được gọi là bộ nhớ. Một vị trí lưu trữ nhất định trong bộ nhớ có thể được liên kết với một hoặc nhiều thiết bị, sao cho chỉ các luồng trên các thiết bị được liên kết mới có quyền truy cập vào. Ngoài ra, mỗi luồng được phép có khung nhìn tạm thời (temporary view) của riêng mình đối với bộ nhớ nói trên. Khung nhìn tạm thời của bộ nhớ cho mỗi luồng không phải là một phần bắt buộc của mô hình bộ nhớ OpenMP, nhưng có thể đại diện cho bất kỳ loại cấu trúc nào, chẳng hạn như thanh ghi máy, bộ đệm hoặc bộ nhớ cục bộ khác, giữa luồng và bộ nhớ. Khung nhìn tạm thời của bộ nhớ cho phép luồng lưu các biến vào bộ nhớ cache và do đó tránh chuyển sang bộ nhớ cho mọi tham chiếu đến một biến duy nhất. Mỗi luồng cũng có quyền truy cập vào một loại bộ nhớ khác mà các luồng khác không được truy cập, được gọi là bộ nhớ riêng của luồng.

Mô hình bộ nhớ này được đáp ứng hoàn toàn khi sử dụng cùng với mô hình đa luồng và thực hiện trên các kiến trúc (một) máy tính sử dụng bộ nhớ đa lõi, vì khi đó, các luồng đều sử dụng chung không gian của tiến trình (lôgic) và không gian nhớ này được cấp phát trên một bộ nhớ vật lý duy nhất của máy tính (RAM). Do đó, các bộ dịch OpenMP hiện thời đều đang được phát triển cho kiến trúc này.

Trong các phần trước của luận án này, chúng ta đã xem xét mô hình hoạt động của CAPE với một mô hình bộ nhớ tương thích ngầm định với chuẩn OpenMP. Chương này chúng ta sẽ quay lại với các vấn đề liên quan đến mô hình bộ nhớ. Một cách tổng quan, vấn đề này bao gồm: 1) Xây dựng mô hình bộ nhớ tương thích với mô hình RC

của OpenMP; 2) Xử lý chỉ thị `flush` của OpenMP; 3) xử lý các mệnh đề dữ liệu chia sẻ khác. Trong [14] đã công bố mô hình bộ nhớ và các đề xuất cho việc xử lý chỉ thị flush và các mệnh đề dữ liệu chia sẻ khác của OpenMP. Luận án này bổ sung thêm trường hợp áp dụng cho CAPE song song hai mức đa luồng.

4.2 Danh sách các chỉ thị và mệnh đề chia sẻ dữ liệu của OpenMP

OpenMP có một tập các mệnh đề liên quan đến dữ liệu chia sẻ. Bảng 4.1 liệt kê danh sách các mệnh đề chia sẻ dữ liệu của OpenMP cùng với mô tả tác dụng của từng mệnh đề. Chi tiết cho các yêu cầu của từng mệnh đề được mô tả rõ trong [1].

Bảng 4.1. Mô tả các loại biến chia sẻ của OpenMP

Mệnh đề	Mô tả
`default (none|shared)`	Chỉ định mặc định tính chất của biến chia sẻ hoặc không.
`shared(list)`	Chỉ định danh sách `(list)` các biến chia sẻ.
`private(list)`	Chỉ định danh sách biến cục bộ.
`firstprivate (list)`	Cho phép chia sẻ giá trị của biến `private` khi vào đoạn vòng lặp song song.
`lastprivate(list)`	Cho phép chia sẻ giá trị của biến `private` khi kết thúc đoạn vòng lặp song song
`copyin(list)`	Cho phép truy cập giá trị của biến `threadprivate`
`copyprivate(list)`	Chỉ định danh sách biến `private` có thể chia sẻ với các luồng khác
`reduction(list, ops)`	Chỉ định danh sách biến `reduction` theo toán tử `ops` để tại kết thúc đoạn vòng lặp để giá trị biến có thể chia sẻ với các luồng khác

Các mệnh đề này có thể được sử dụng trong các chỉ thị OpenMP, với danh sách được liệt kê như trong Bảng 4.2. [2]

Bảng 4.2. Các chỉ thị có thể sử dụng các mệnh đề chia sẻ dữ liệu trong OpenMP

	parallel	for	sections	single	parallel for	parallel sections
private	✓	✓	✓	✓	✓	✓
firstprivate	✓	✓	✓	✓	✓	✓
lastprivate		✓	✓		✓	✓
shared	✓				✓	✓
default	✓				✓	✓
copyin	✓				✓	✓
copyprivate				✓		
reduction	✓	✓	✓		✓	✓

Ví dụ sau minh họa cho một chỉ thị OpenMP có sử dụng mệnh đề dữ liệu chia sẻ.

```
…
#pragma omp parallel private(i) firstprivate(j)
{   i = 3;   j = j + 2;
   assert (*ptr_i == 1 && *ptr_j == 2);
}
…
```

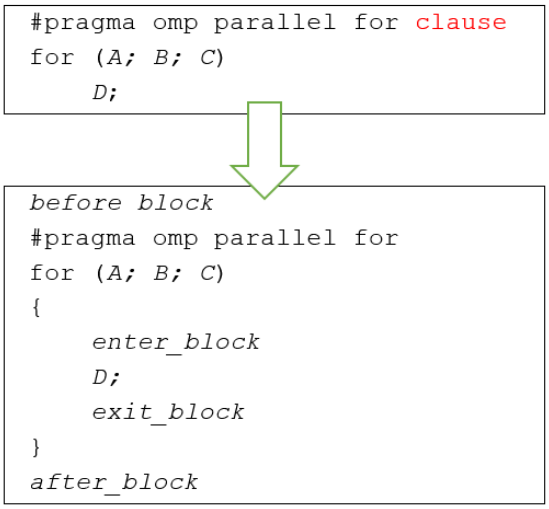
Trong đó, mệnh đề `private(i)` chỉ định biến `i` là cục bộ cho mỗi luồng thực hiện công việc song song, tức là mỗi luồng sẽ sử dụng một biến `i` cục bộ riêng của mình, không liên quan đến biến `i` đã tồn tại trước đó; `firstprivate(j)` chỉ định đặc tính tương tự cho biến `j`, nhưng có thêm một yêu cầu là các biến `j` cục bộ sẽ được khởi tạo với giá trị của biến `j` đã tồn tại trước đó.

4.3 Xử lý các chỉ dẫn chia sẻ dữ liệu của OpenMP trên CAPE

4.3.1 Nguyên tắc xử lý

Nguyên tắc xử lý các mệnh đề chia sẻ của OpenMP trên CAPE đã được đề xuất trong [14], bằng cách xây dựng các khuôn dạng chuyển đổi mới hoặc bổ sung vào các khuôn dạng chuyển đổi đã có các câu lệnh để thực hiện các mệnh đề chia sẻ dữ liệu.

Ví dụ: khuôn dạng chuyển đổi cho cấu trúc `parallel for` - là cấu trúc điển hình và căn bản nhất cho việc biên dịch chương trình OpenMP của CAPE [15] - sẽ được bổ sung thêm các khối lệnh: *`before_block`*, *`enter_block`*, *`exit_block`* và *`after_block`* để xử lý với các mệnh đề chia sẻ dữ liệu `clause_U` như trong hình dưới đây.



Hình 4.2. Khuôn dạng tổng quát việc chuyển đổi cấu trúc `parallel for` với các mệnh đề dữ liệu chia sẻ

Theo khuôn dạng tổng quát này, chương trình dịch khởi tạo tất cả các khối *before_block*, *enter_block*, *exit_block* và *after_block* với giá trị rỗng. Sau đó duyệt tuần tự các mệnh đề chia sẻ dữ liệu để thêm vào từng phần phù hợp.

4.3.2 Các khuôn dạng chuyển đổi cụ thể cho các mệnh đề

4.3.2.1 Khuôn dạng chuyển đổi cho cấu trúc `parallel for`

Các câu lệnh bổ sung cụ thể cho từng mệnh đề chia sẻ của OpenMP được liệt kê trong Bảng 4.3, ngoại trừ mệnh đề `reduction` và `copyprivate` được xử lý trong các mục tiếp theo.

Bảng 4.3. Ánh xạ các hàm của CAPE tương ứng các mệnh đề chia sẻ của OpenMP

		1	2	3	4
		before_block	*enter_block*	*exit_block*	*after_block*
1	shared (x)				
2	threadprivate (x)	typeof (x) __x__ ;	typeof (__x__) x;		
3	private (x)	typeof (x) __x__;	typeof (__x__) x;		
4	firstprivate (x)	typeof (x) __x__; __x__ = x;	typeof (__x__) x; x = __x__		
5	lastprivate (x)	typeof (x) __x__ ;	typeof (__x__) x ;	if (thread_num == (num_threads - 1)) __x__ = x	x = __x__ ;

		1	2	3	4
		before_block	*enter_block*	*exit_block*	*after_block*
6	reduction (x)	typeof (x) __x__;	typeof (__x__) x; init_value (x) ;	send (x, master);	tiếp tục ở mục 4.3.2.2

7	copyin (x)	typeof (x) __x__ ;
		if (master ( ))
		__x__ = x ;
			if (master ( )) {
			x = __x__ ;
			broadcast (x) ;
			}
			else {
			receive (x); inject (x) ;
			}		

8	copyprivate	Chi tiết ở mục 4.3.2.3	Chi tiết ở mục 4.3.2.3	Chi tiết ở mục 4.3.2.3	Chi tiết ở mục 4.3.2.3

4.3.2.2 Mệnh đề `reduction`

Mệnh đề này yêu cầu tiến trình chính phải kết xuất dữ liệu tổng hợp từ các dữ liệu thành phần do các tiến trình phụ tính toán được. Để giải quyết việc chuyển đổi mệnh đề `reduction` trên CAPE có 2 giải pháp thực hiện: 1) tích hợp giá trị của biến cần kết xuất trên ảnh chụp tiến tình của từng nút phụ rồi gửi về nút chính để tổng hợp; và 2) sử dụng một hàm riêng cho biến cần kết xuất để gửi và nhận giá trị biến này trên cả nút phụ và nút chính.

```
0    init ( x ) ;
1    if ( master ( ) ) {
...
8          stop ( ) ;
8a         init ( update_list ) ;
9          wait_for ( after ) ;
9a         merge ( update_list, after ) ;
9b         receive_reduction ( x ) ;
9c         merge ( x, after ) ;
...
15   } else {
...
22         send ( after_i , master ) ;
22a        send ( x , master ) ;
```

Hình 4.3. Khuôn mẫu chuyển đổi của CAPE cho mệnh đề reduction

Ví dụ, để thực hiện mệnh đề `reduction` cho biến `x`, theo khuôn dạng này, một hàm mới `init(x)` ở dòng thứ 0 sẽ được chạy trên tất cả các nút để thực hiện khởi tạo biến `x`. Mỗi nút phụ, sau khi thực hiện phần việc của mình, gửi kết quả của công việc thực hiện được theo dạng một ảnh chụp bình thường (dòng 22), sẽ gửi thêm một giá trị của biến `x` tới nút chính (dòng 22a). Đối với nút chính, sau khi nhận kết quả từ nút phụ, hàm `receive_reduction(x)` (dòng 9b) sẽ nhận tất cả giá trị biến `x` từ các nút phụ đồng thời thực hiện kết xuất giá trị tổng hợp vào biến `x` ở nút chính. Sau đó giá trị của biến `x` được thêm vào ảnh chụp tiến trình (dòng 9c) trước khi ảnh này được gửi tới tất cả các slave để thực hiện việc đồng bộ hóa không gian nhớ của tất cả các nút.

4.3.2.3 Mệnh đề copyprivate

`copyprivate` yêu cầu câu lệnh phải thực hiện trong vùng `single`. Thông thường, phần `single` được chạy ở nút chính, do vậy hàm `broadcast()` tại đoạn kết thúc của khối `single` sẽ gửi giá trị biến cập nhật nhất đến tất cả các nút phụ. Nhận được giá trị cần cập nhật, các nút phụ cập nhật vào vùng nhớ của tiến trình.

```
# pragma omp single copyprivate ( x )
    D ;
```

↓ automatically translated into ↓

```
# pragma omp single
{
    D ;
    broadcast ( x ) ;
}
if ( !master ( ) )
{
    receive ( x ) ;
    inject ( x ) ;
}
```

Hình 4.4. Khuôn mẫu cho mệnh đề `copyprivate`

4.4 Giải thích của cơ chế chuyển đổi các mệnh đề dữ liệu chia sẻ của OpenMP trên CAPE

Trong phần này sẽ giải thích tính logic các khuôn dạng chuyển đổi chỉ dẫn chỉ chia sẽ dữ liệu của OpenMP trên CAPE đã đảm bảo được yêu cầu chia sẽ dữ liệu của OpenMP, đồng thời cần thực nghiệm kết quả chạy chương trình OpenMP trên các hệ thống sử dụng bộ nhớ chia sẻ và trên CAPE phải giống nhau. Điều kiện là cần phải có hai giả thiết dưới đây.

Giả thiết thứ nhất: chương trình theo dõi (monitor/checkpointer) chạy chính xác và không làm ảnh hưởng đến cấu trúc dữ liệu hoặc việc gọi hàm và kết quả của chương trình chạy. Nghĩa là chạy một chương trình OpenMP nguyên thủy trên một máy tính và chạy cùng chương trình đó ở dạng đã chuyển đổi sang CAPE – có sử dụng chụp ảnh tiến trình - đều cho kết quả giống nhau. Thực tế là chương trình giám sát phải theo dõi chương trình chạy để kiểm soát được các vùng nhớ bị thay đổi và đồng thời có thể cập nhật lại giá trị biến của chương trình chạy nhưng không làm thay đổi cấu trúc và logic của chương trình chạy. Trong các khuôn dạng chuyển đổi, chương trình monitor có bổ sung vào các hàm `create()`, `stop()`, `merge()`, `wait_for()`,... tuy nhiên không làm thay đổi cấu trúc dữ liệu của chương trình đang chạy. Vì vậy, trình giám sát không làm thay đổi kết quả của trình ứng dụng.

Giả thiết thứ hai: đoạn chương trình chạy song song nếu không có chỉ thị riêng, chẳng hạn như chỉ thị `reduction` đồng thời nếu không thỏa mãn điều kiện Bernstein [63] thì trên cùng một địa chỉ vùng nhớ chia sẻ (hay giá trị của biến chia sẻ) luồng chương trình nào cập nhật cuối cùng sẽ cập nhật giá trị cuối cùng của các biến.

4.4.1 Trường hợp 1: Đoạn chương trình song song thỏa mãn điều kiện Bernstein

Gọi I_i là tập các biến được đoạn chương trình P_i (trên nút i) đọc khi chạy và O_i là tập các biến được đoạn chương trình P_i ghi. Trong ngữ cảnh của CAPE, một biến được hiểu là một vùng nhớ. Theo điều kiện của Bernstein, P_1 và P_2 có thể chạy đồng thời khi và chỉ khi ba điều kiện sau đồng thời thỏa mãn: $I_1 \cap O_2 = I_2 \cap O_1 = O_1 \cap O_2 = \emptyset$. Hai điều kiện đầu của Bernstein với ý nghĩa rằng P_1 và P_2 có thể thực hiện chương trình độc lập mà không cần phải yêu cầu một đoạn chương trình phải chạy trước để cung cấp dữ liệu cho chương trình kia. Điều kiện thứ 3 của Bernstein với ý nghĩa rằng không có sự cập nhật dữ liệu chia sẻ của P_1 và P_2. Đồng nghĩa rằng ảnh chụp tiến trình tại nút

chính có thể được cập nhật từ ảnh chụp tiến trình của P_1 và P_2 theo bất kỳ thứ tự nào. Cụ thể, ảnh bộ nhớ các biến của đoạn chương P_i trước và sau khi chạy sẽ có thay đổi gọi là $delta_i$ lưu tất cả các biến được đoạn chương trình P_i cập nhật giá trị mới. File $delta_i$ đại diện cho các O_i của P_i. Tại nút chính, sau khi nhận được các ảnh bộ nhớ của các P_i được gộp lại vào vùng nhớ của nó trước khi gửi ảnh bộ nhớ mới này đến các P_i cho các đoạn mã xử lý song song tiếp theo. Theo điều kiện thứ 3 của Bernstein, không có biến chia sẻ nào chung giữa các P_i đồng nghĩa là các vùng nhớ (biến chia sẻ) bị thay đổi trên các P_i luôn khác P_j với $i \neq j$. Do vậy, việc gộp các ảnh vùng nhớ của các P_i vào ảnh vùng nhớ của nút chính như gộp các mảnh ghép rời nhau của một bức tranh. Cũng theo điều kiện 1 và 2 của Bernstein, P_1 và P_2 độc lập không yêu cầu chương trình nào cần phải chạy trước nên thứ tự gộp ảnh bộ nhớ vào ảnh bộ nhớ của nút chính đều như nhau.

4.4.2 Trường hợp 2. Đoạn chương trình song song không thỏa mãn điều kiện Bernstein

Trường hợp 2.1: Không có thêm chỉ thị riêng của OpenMP.

Trong trường hợp này sẽ có việc truy cập hoặc cập nhật biến chia sẻ chung giữa các P_i, trong ngữ cảnh của CAPE là cập nhật giá trị cùng một vùng địa chỉ giống nhau, tại nút chính một phần ảnh bộ nhớ của P_i bị trùng địa chỉ với P_j sẽ được giá trị biến tại vùng nhớ trùng nhau này được ghép thay thế bởi P_j gửi ảnh tiến trình đến sau như vậy thỏa mãn theo giả thiết thứ 2, kết quả được trả về theo P_i nào có cập nhật gần nhất.

Trường hợp 2.2: Có thêm chỉ thị riêng của OpenMP.

Với nguyên lý của CAPE xử lý việc cập nhật ảnh vùng nhớ của hai P_1 và P_2 có phần trùng nhau sẽ lấy theo kết quả ảnh vùng nhớ trùng nhau của P_i gửi đến sau. Trong trường hợp chương trình có các chỉ thị như `reduction`, kết quả chương trình sẽ sai vì không tổng hợp được kết quả từng P_i thành kết quả cuối cùng theo một toán tử nào đó của `reduction`. Để thực hiện mệnh đề `reduction`, CAPE tổ chức thêm hàm khởi tạo `init(x)` ở dòng 0 trong Hình 4.3 và các P_i gửi riêng giá trị biến `x` (dòng 22a) đến nút chính sau khi các ảnh vùng nhớ của từng chương trình P_i gửi về nút chính để tổng hợp như trường hợp 1 và trường hợp 2.1. Dòng 9b, 9c trong Hình 4.3 thực hiện việc tổng hợp biến `x` theo phương pháp `reduction` theo toán tử của chương trình (ví dụ: toán tử + sẽ cộng dồn giá trị của biến x_i của từng P_i) sau đó cập nhật giá trị nào vào

đúng vị trí ảnh vùng nhớ của nút chính trước khi ảnh vùng nhớ này có thể gửi đến các nút phụ để thực hiện các đoạn mã song song tiếp theo.

Đối với mệnh đề `shared(x)`, hàng 1 trong Bảng 4.3 có ý nghĩa biến `x` được chia sẻ cho các slave, CAPE áp dụng nguyên lý P_i nào cập nhật cuối cùng sẽ là giá trị kết quả của biến x, tương ứng với trường hợp 2.1.

Đối với mệnh đề `threadprivate(x), private(x), firstprivate(x), lastprivate(x)`, tại dòng 2 đến 5 trong Bảng 4.3 biến `x` được copy bằng cách khai báo bên đoạn mã của vòng lặp để tạo ra biết cục bộ trong vòng lặp. Đương nhiên biến `x` cục bộ này sẽ được cấp phát trên vùng nhớ có địa chỉ khác với biến `x` ngoài vòng lặp, riêng với `firstprivate (x)`, biến `x` cục bộ được khởi tạo giá trị ban đầu bằng biến `x` bên ngoài vòng lặp thông qua biến trung gian `__x__`. Ngược lại, đối với `lastprivate(x)` giá trị của biến `x` cần được lưu giữ sau khi kết thúc vòng lặp thông qua biến trung gian `__x__`, đồng thời cần lấy giá trị `x` của P_i có cập nhật cuối cùng như mô phỏng trong dòng 5 cột 3 của Bảng 4.3. Trong quá trình CAPE thực hiện tạo ảnh vùng nhớ có thay đổi gửi về nút chính vùng nhớ của biến `x` cục bộ vẫn được ghi nhận khác địa chỉ với vùng nhớ của biến `x` bên ngoài vòng lặp nên không ảnh hưởng đến giá trị của x ngoài vòng lặp, tương ứng thỏa mãn trường hợp 2.1.

Đối với mệnh đề `copyin(x), copyprivate(x)`, dòng 7 và 8 trong Hình 4.4 sử dụng hàm `broadcast(x)` để cập nhật giá trị x đến các P_i. Các P_i nhận được giá trị `x` và tích hợp vào không gian nhớ của mình để tiếp tục thực hiện phần còn lại của chương trình. Trong quá trình chạy chương trình nếu P_i có thực hiện cập nhật giá trị của biến `x` sẽ được ghi nhận trong ảnh chụp vùng nhớ của P_i đồng thời gửi đến nút chính tập hợp theo như kịch bản trường hợp 2.1 thỏa mãn giả thiết 2.

Như vậy với trường hợp 2.2: chỉ có trường hợp mệnh đề `reduction` cần phải tổ chức xử lý riêng bằng hàm mới, đối với các mệnh đề, chỉ thị chia sẻ khác đều có thể dẫn về trường hợp 2.1 thỏa mãn giả thiết 2.

Đối với phiên bản CAPE đa luồng được đề cập trong luận án này có thêm điểm mới là các nút sẽ chạy đa luồng tuy nhiên như đã đề cập trong 2.2, phần đa luồng trên từng nút sử dụng lại mã của OpenMP và không can thiệp đến vùng nhớ của đoạn mã

này nên theo logic giải thích được đề cập trong phần này, các trường hợp cần giải thích ở trên là đủ.

Về cài đặt chương trình CAPE thực hiện các mệnh đề chia sẻ dữ liệu của OpenMP trên hệ thống bộ nhớ phân tán, nhóm nghiên cứu đã cài đặt thành công các mệnh đề `threadprivate(x)`, `private(x)`, `firstprivate(x)`, `lastprivate(x)`, `copyin(x)`, `copyprivate(x)`, `reduction` với kết quả chạy chương trình giống với chương trình gốc OpenMP. Toàn bộ chương trình được công bố dưới dạng mã nguồn mở GitHub (https://github.com/) truy cập trực tuyến như ở PHỤ LỤC 1.

4.5 Tiểu kết Chương IV

Chương này đã trình bày cách thức xử lý các chỉ thị chia sẻ dữ liệu của OpenMP trên CAPE. Nói cách khác thực hiện việc chia sẽ đồng bộ dữ liệu ảnh chụp tiến trình của CAPE như thế nào để đáp ứng được các chỉ thị chia sẻ dữ liệu của OpenMP. Qua phân tích và thực nghiệm, CAPE có thể xử lý được tất cả các mệnh đề dữ liệu chia sẻ của OpenMP với các ưu điểm: Thứ nhất, người lập trình không yêu cầu phải tự tổ chức gửi nhận đồng bộ dữ liệu của các đoạn chương trình tuần tự hoặc song song. Thứ hai, số nút xử lý song song có thể tăng giảm linh động theo nhu cầu (không có giới hạn số lượng ảnh chụp tiến trình gửi về nút chính).

Các kết quả của chương này được công bố ở công trình [CT3].

KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN CỦA LUẬN ÁN

KẾT LUẬN

OpenMP, với những ưu điểm về tính đơn giản, dễ học, dễ sử dụng, khả năng cung cấp hiệu năng cao, vẫn tiếp tục được các nhà nghiên cứu tìm cách chuyển đổi lên các hệ thống sử dụng bộ nhớ phân tán. CAPE là một tiếp cận nằm trong xu thế đó, với những kết quả về lý thuyết cũng như thực nghiệm ban đầu đã chứng minh được tiềm năng lớn trong tính tương thích hoàn toàn với OpenMP cũng như hiệu năng cao khi thực hiện các chương trình OpenMP trên các hệ thống máy tính phân tán.

Luận án đã thành công trong việc mở rộng mô hình hoạt động của CAPE trước đấy để khai thác tốt hơn khả năng của các bộ vi xử lý đa lõi trên các nút tính toán. Mô hình mới đã bổ sung thêm một mức song song khi thực hiện các đoạn mã tính toán trên các nút này, làm cho việc thực hiện một câu lệnh song song `S` dạng OpenMP được tiến hành song song theo 2 mức: 1) Các nút tính toán thực hiện song song từng phần S_i của `S` trên các nút tính toán (i đại diện cho số nút tính toán); và 2) Từng phần S_i tại mỗi nút được thực hiện một cách song song theo dạng 1 tiến trình đa luồng OpenMP. Các kết quả phân tích lý thuyết cũng như những kết quả thực nghiệm đã chứng minh mô hình mới đã giúp hiệu năng hệ thống khi áp dụng CAPE tăng tuyến tính với số lõi của các bộ xử lý trên các nút tính toán.

Việc cài đặt mô hình thực hiện mới đòi hỏi phải xây dựng lại công cụ căn bản và quan trọng nhất của CAPE là công cụ chụp ảnh tiến trình. Sự thay đổi so với công cụ trước đây là rất lớn, mang tính cốt lõi, để có thể xử lý được việc chụp được ảnh các tiến trình đa luồng tại các nút tính toán. Luận án đã phát triển thành công kỹ thuật chụp ảnh tiến trình phù hợp với CAPE đa luồng, chạy ở mức không gian người sử dụng, thay vì chạy ở không gian nhân của hệ điều hành như trước đây giúp có ưu điểm là tính ổn định cao với các phiên bản khác nhau của OS do chỉ gọi lại các hàm thư viện phổ thông của OS cung cấp.

Kết quả của luận án đã đạt được bao gồm:

1. Phân tích và thực nghiệm các hướng mở rộng mô hình CAPE trên hệ thống máy tính CPU đa lõi và chọn giải pháp khả thi để thực hiện [CT1] [CT2].

2. Tổng hợp và phân loại các kỹ thuật chụp ảnh tiến trình- kỹ thuật nền tảng lõi áp dụng cho CAPE- từ đó đề xuất áp dụng kỹ thuật chụp ảnh tiến trình phù hợp cho CAPE đa luồng [CT4]

3. Đề xuất mô hình hoạt động mới CAPE đa luồng, xây dựng và thực nghiệm thành công hệ thống phần mềm CAPE đa luồng áp dụng trên hệ thống tính toán đa lõi với hiệu năng có thể so sánh được với MPI - công cụ có khả năng cung cấp hiệu năng cao nhất trên các hệ thống này [CT5].

4. Xử lý được các vấn đề chia sẻ dữ liệu của CAPE đa luồng [CT3].

HƯỚNG PHÁT TRIỂN LUẬN ÁN

Từ những kết quả đạt được trong luận án một số vấn đề cần được quan tâm nghiên cứu trong thời gian tới:

1. Mô hình hoạt động của CAPE có được phát triển tiếp tục theo hướng khai thác khả năng song song của các card đồ họa.

2. Nếu được một đơn vị/nhóm phát triển phần mềm hệ thống quan tâm đầu tư và tiếp tục phát triển một cách nghiêm túc, CAPE hoàn toàn có khả năng trở thành một công cụ tốt để đưa OpenMP lên các hệ thống bộ nhớ phân tán, tương đương như MPI, mang lại một giải pháp lập trình song song dễ học, dễ lập trình và có hiệu năng cao trên các hệ thống này. Chúng tôi cũng đã đưa mã nguồn của phần mềm CAPE lên Github – kho mã nguồn mở của cộng đồng – với định hướng cung cấp mã nguồn CAPE để cộng đồng cùng tiếp tục cùng nghiên cứu, phát triển.

DANH MỤC CÁC CÔNG TRÌNH LIÊN QUAN ĐẾN LUẬN ÁN

[CT1]. **Đỗ Xuân Huyền**, Hà Viết Hải (2018), "Các giải pháp mở rộng mô hình hoạt động của CAPE cho mạng máy tính sử dụng bộ vi xử lý đa lõi". *Tạp chí Khoa học và Công nghệ (Trường Đại học Khoa học - Đại học Huế)*, ISSN 2354-0842, Tập 12, Số 1, 2018, Tr. 51–61. http://joshusc.hueuni.edu.vn/jos_articles.php?article_id=385.

[CT2]. Hà Viết Hải, **Đỗ Xuân Huyền** (2018), "Mở rộng mô hình hoạt động của CAPE bằng sử dụng máy ảo". *Tạp chí Tạp chí Khoa học Đại học Huế: Kỹ thuật và Công nghệ*; ISSN 1859–1388, Tập 127, Số 2A, 2018, Tr. 159–168. http://jos.hueuni.edu.vn/index.php/hujos-tt/article/view/4795.

[CT3]. **Đỗ Xuân Huyền**, Hà Viết Hải, Trần Văn Long (2019), "Xử lý các mệnh đề về dữ liệu chia sẻ của OpenMP trên các hệ thống sử dụng bộ nhớ phân tán". *Kỷ yếu Hội nghị khoa học quốc gia lần thứ XII về Nghiên cứu cơ bản và ứng dụng Công nghệ thông tin (FAIR)*. Huế 2019, Tr. 577–583. https://doi.org/10.15625/vap.2019.00074.

[CT4]. **X. H. Do**, V. H. Ha, V. L. Tran and É. Renault, "The technique of locking memory on Linux operating system - Application in checkpointing," 2019 6th NAFOSTED Conference on Information and Computer Science (NICS), 2019, pp. 178-183. https://doi.org/10.1109/NICS48868.2019.9023816.

[CT5]. **Xuan Huyen Do**, Viet Hai Ha, Van Long Tran, Eric Renault (2021), "New execution model of CAPE using multiple threads on multi-core clusters", *ETRI Journal (SCIE)*, May, 2021. https://doi.org/10.4218/etrij.2020-0201.

TÀI LIỆU THAM KHẢO

[1] OpenMP Architecture Review Board (2018), "OpenMP specification 5.0", Available: *https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf* (last accessed 5th Oct 2019).

[2] OpenMP Architecture Review Board (2015), "OpenMP Application Program Interface v4.5", Technical Report. *https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf*, pp 17, 2015.

[3] OpenMP.org, "OpenMP Compilers & Tools", Available: *https://www.openmp.org/resources/openmp-compilers-tools* (last accessed 10th Aug 2020).

[4] MPI Forum (2018), "Message Passing Interface Forum", Available: *https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf* (last accessed 6th Aug 2021).

[5] IDG Communications, Inc. (2015). "Intel pushes 10nm chip-making process to 2017", slowing Moore's Law, Available: *https://www.infoworld.com/article/2949153/intel-pushes-10nm-chipmaking-process-to-2017-slowing-moores-law.html* (last accessed 6th Aug 2023).

[6] D. Margery, G. Vallée, R. Lottiaux, C. Morin, J. Berthou (2003). Kerrighed: "A SSI Cluster OS Running OpenMP", *EWOMP 2003*, Fifth European Workshop on OpenMP.

[7] M. Sato, H. Harada, A. Hasegawa and Y. Ishikaw (2001), "Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software dis-tributed shared memory system", *Journal Scientific Programming*, Volume 9 Issue 2,3, 2001.

[8] A. Saa-Garriga, D. Castells-Rufas, J. Carrabina (2015). "OMP2MPI: Automatic MPI code generation from OpenMP programs", Proceedings of the *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES) 2015*. Amsterdam, January 21st. Collocated with HIPEAC 2015 Conference.

[9] Jacob A.C. et al. (2015). "Exploiting Fine- and Coarse-Grained Parallelism Using a Directive Based Approach". OpenMP: Heterogenous Execution and

Data Movements. *Lecture Notes in Computer Science*, vol 9342. Springer, Cham, pp. 30-41.

[10] L. Huang and B. Chapman and Z. Liu (2005). "Towards a more efficient implementation of OpenMP for clusters via translation to Global Arrays", *Journal of Parallel Computing 31*, pp 1114–1139.

[11] J.P Hoeflinger (Intel) (2010). "Cluster OpenMP* for Intel® Compilers", Available: *https://software.intel.com/en-us/articles/cluster-openmp-for-intel-compilers*, (last accessed 15th Aug 2017).

[12] Viet Hai Ha and Eric Renault (2011). "Discontinuous Incremental: A new approach towards extremely lightweight checkpoints". In *Computer Networks and Distributed Systems (CNDS), 2011 International Symposium on*. IEEE, 227–232.

[13] Viet Hai Ha and Éric Renault (2011). "Improving performance of CAPE using discontinuous incremental checkpointing". In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th Internaotional Conference on*. IEEE, 802– 807.

[14] Viet Hai Ha and Éric Renault (2012). "Design of a shared-memory model for CAPE", In *The 8th International Workshop on OpenMP (IWOMP 2012)*, Rome, Italy, Springer LNCS 7321, pp. 262-266, 2012.

[15] Viet Hai Ha and Éric Renault (2011). "Design and performance analysis of CAPE based on discontinuous incremental checkpoints", In *2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 2011.

[16] Van Long Tran, Éric Renault, Viet Hai Ha (2015). "Improving the Reliability and the Performance of CAPE by Using MPI for Data Exchange on Network", *International Conference on Mobile, Secure and Programmable Networking (MSPN'2015). Paris, France, 6/2015. Mobile, Secure, and Programmable Networking. Volume 9395 of the Springer series Lecture Notes in Computer Science pp. 90-100.* https: //doi.org/*10.1007/978-3-319-25744-0_8.*

[17] Van Long Tran, Éric Renault, Viet Hai Ha (2016). "Analysis and evaluation of the performance of CAPE", In *The 16th IEEE International Conference on Scalable Computing and Communications*. Toulouse, France, 2016.

[18] Van Long Tran , Éric Renault, Xuan Huyen Do, Viet Hai Ha (2017). "Design and implementation of a new execution model for CAPE", *Proceedings of the Eighth International Symposium on Information and Communication Technology* (SoICT's 2017), ACM, pp. 453–459.

[19] Van Long Tran, Éric Renault, Viet Hai Ha, Xuan Huyen Do (2018). "Time-stamp Incremental Checkpointing and its Application for an Optimization of Execution Model to Improve Performance of CAPE", *Informatica 42* (2018) 301–311.

[20] Hà Viết Hải và Trần Văn Long (2013). "Phân tích và đánh giá hiệu năng của CAPE". *Kỷ yếu hội thảo Quốc gia về Nghiên cứu và Ứng dụng Công nghệ thông tin*, 20-21 tháng 6/2013, Huế, Việt Nam (FAIR 2013).

[21] OpenSSI Project, "OpenSSI (Single System Image). Clusters for Linux", Available: *http://openssi.org/cgi-bin/view?page=openssi.html*, (last accessed 10th Mar 2017).

[22] Omni Compiler Project. Available: *http://omni-compiler.org/* (last accessed 11th Aug 2017).

[23] J. Lee, M. Sato (2010). "Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems", *39th International Conference on Parallel Processing*, ICPP Workshops 2010, San Diego, California, USA.

[24] J. Tao, W. Karl, C. Trinitis (2005). "Implementing an OpenMP Execution Environment on InfiniBand Clusters", *In proceeding of the First International Workshop on OpenMP (IWOMP 2005)*. Eugene, Oregon.

[25] InfiniBand Trade Association (2003). "InfiniBand Architecture Specification", Volume 2, Release 1.1.

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

[27] Nieplocha, J., Krishnan, M., Palmer, B., Tipparaju, V., Harrison, R., Chavarría-Miranda, D. (2011). “Global Arrays Parallel Programming Toolkit”. In: Padua, D. (eds) *Encyclopedia of Parallel Computing.* Springer, Boston, MA.

[28] Lei Huang, Barbara Chapman, Zhenying Liu and Ricky Kendall (2004). “Efficient Translation of OpenMP to Distributed Memory”. *Lecture Notes in Computer Science*, 2004, Volume 3038/2004, 408-413.

[29] Lei Huang, Barbara Chapman and Ricky Kendall (2003). “OpenMP for Clusters”. *In the Fifth European Workshop on OpenMP*, EWOMP’03, Aachen, Germany, 2003.

[30] Robert Lyerly, Sang-Hoon Kim, and Binoy Ravindran (2019). “LibMPNode: An OpenMP Runtime For Parallel Processing Across Incoherent Domains”. *In Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'19)*. Association for Computing Machinery, New York, NY, USA, 81–90. https://doi.org/10.1145/3303084.3309495.

[31] D. Katz, A. Barbalace, S. Ansary, A. Ravichandran, and B. Ravindran (2015). “Thread Migration in a Replicated-Kernel OS”. *In 2015 IEEE 35th International Conference on Distributed Computing Systems*. 278–287. https://doi.org/10.1109/ICDCS.2015.36.

[32] Sang-Hoon Kim, Robert Lyerly, and Pierre Olivier (2017). “Popcorn Linux: Compiler, Operating System and Virtualization Support for Application/Thread Migration in Heterogeneous ISA Environments”. *Presented at the 2017 Linux Plumbers Conference*. http://www.linuxplumbersconf.org/2017/ocw/proposals/4719.html (last accessed 03th Jun 2023).

[33] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia (2013). “A page coherency protocol for popcorn eplicatedkernel operating system”. *In Proceedings of the ManyCore Architecture Research Community Symposium (MARC)*.

[34] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araujo (2022). "The OpenMP Cluster Programming Model". *In Workshop Proceedings of the 51st International Conference on Parallel Processing (ICPP Workshops '22).* Association for Computing Machinery, New York, NY, USA, Article 17, 1–11. https://doi.org/10.1145/3547276.3548444

[35] James S.Plank (1997). "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance", *Technical Report UT-CS-97-372*, University of Tennessee.

[36] G. M. Amdahl (1967), "Validity of the single-processor approach to achieving large scale computing capabilities", Available: *http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf* (last accessed 10th Aug 2019).

[37] Plank, James S and Beck, Micah and Kingsley, Gerry and Li, Kai (1994). "Libckpt: Transparent checkpointing under unix", *White Paper, Computer Science Department*.

[38] Cores, Iv´an and Rodr´ıguez, M´onica and Gonz´alez, Patricia and Mart´ın, Mar´ıa J (2016). "Reducing the overhead of an MPI application-level migration approach", *Parallel Computing*, Elsevier, pp. 72–82

[39] Z. Chen, J. Sun, and H. Chen (2016), "Optimizing checkpoint restart with data eduplication," *Scientific Programming*, vol. 2016.

[40] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira (2004). "Adaptive incremental checkpointing for massively parallel systems". *In ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*, pages 277–286, St. Malo, France, 2004. ACM.

[41] J. Duell (2005). "The design and implementation of berkeley lab's linux checkpoint/restart," *Lawrence Berkeley National Laboratory*, 2005.

[42] Luciano A. Stertz (2003). "Readable dirty-bits for ia64 linux". *Internal requirement specification, HewlettPackard*, 2003.

[43] J. Ansel, K. Aryay, and G. Coopermany (2009). "Dmtcp: Transparent checkpointing for cluster computations and the desktop", *in Parallel & Distributed Processing*, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009, pp. 1–12.

[44] Parallels; OpenVZ, "Checkpoint/Restore In Userspace", *Open Source Software Community*, Available: *http://criu.org/*. (last accessed 12th Aug 2019).

[45] Linus Torvalds (2013). "Linux kernel source tree". Available: *https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/COPYING* (last accessed 10th Aug 2018).

[46] Mel Gorman (2004). "Understanding the Linux Virtual Memory Manager", *Prentice Hall PTR Upper Saddle River*, NJ, USA ©2004, ISBN:0131453483. pp. 33-34.

[47] Intel Inc. 5-Level Paging and 5-Level EPT (2017). "White Paper. Revision 1.1, Available: *https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf* (last accessed 17th Aug 2019).

[48] J. Corbet (2017). "Linux info from the source", Available: *https://lwn.net/Articles/717293/* (last accessed 25th Feb 2019).

[49] Kirill A. Shutemov, "Linux info from the source", 8 Dec 2016. Available: *https://lwn.net/Articles/708526/* (last accessed 10th Mar 2019).

[50] Li, C-CJ and Fuchs, W Kent (1990). "Catchcompiler-assisted techniques for checkpointing", *Fault-Tolerant Computing (FTCS)*, IEEE, pp. 74–81.

[51] Plank, James S and Xu, Jian and Netzer, Robert HB (1995). "Compressed differences: An algorithm for fast incremental checkpointing", *Technical Report CS-95*-302, University of Tennessee.

[52] Hyochang, NAM and Jong, KIM and Hong, Sung Je and Sunggu, LEE (2002). Probabilistic checkpointing, *IEICE TRANSACTIONS on Information and Systems*, The Institute of Electronics, Information and Communication Engineers, pp. 1093–1104.

[53] Mehnert-Spahn, John and Feller, Eugen and Schoettner, Michael (2009). "Incremental checkpointing for grids", *Linux Symposium*, Montreal, Quebec, Canada, pp. 201–220

[54] Cores, Iv´an and Rodr´ıguez, Gabriel and Gonz´alez, Patricia and Osorio, Roberto R and others (2013). "Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes", *New Generation Computing, Springer*, pp. 163–185.

[55] Red Hat. "CRIU - Checkpoint/Restore in user space". Available : *https://access.redhat.com/articles/2455211* (last accessed 10th May 2019).

[56] I. Ljubuncic, R. Giri, A. Rozenfeld, and A. Goldis (2014). "Be Kind, Rewind — Checkpoint & Restore Capability for Improving Reliability of Large-scale Semiconductor Design", *IEEE HPEC-14*, Sept., 2014.

[57] Pradeep Padala (2002). "Playing with ptrace, Part I", *Linux Journal archive*, Volume 2002 Issue 103, November 2002. Available: *https://www.linuxjournal.com/article/6100.* (last accessed 10th Oct 2019).

[58] Pradeep Padala (2002). "Playing with ptrace, Part II", *Linux Journal archive*, Volume 2002 Issue 104, December 2002. Available: *https://www.linuxjournal.com/article/6210.* (last accessed 10th Oct 2019).

[59] Michael Kerrisk. Linux man pages online. Available: *http://man7.org/linux/man-pages/man2/mprotect.2.html* (last accessed 10th Oct 2017).

[60] Linus Torvalds (2023), "Source code Linux kernel: /mm/mprotect.c", *https://github.com/torvalds/linux/blob/master/mm/mprotect.c* (last accessed 28th Jul 2023).

[61] Dorta, Antonio J and Badía, José M and Quintana, Enrique S and de Sande, Francisco (2005). "Implementing OpenMP for clusters on top of MPI", *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer*, pp. 148–155.

[62] Canonical Ltd, "Ubuntu manpage: pthread_create - create a new thread", Available: *http://manpages.ubuntu.com/manpages/bionic/man3/pthread_create.3.htm* (last accessed 25th Oct 2019).

[63] Bernstein (1966). “Analysis of Programs for Parallel Processing”, *IEEE Transaction on Electronic Computers*, IEEE, pp. 757–763.

[64] Hà Viết Hải, Nguyễn Cảnh Hoài Đức, Đỗ Xuân Huyền (2016). “Sử dụng nhiều tiến trình trên các nút tính toán để gia tăng hiệu năng hoạt động của CAPE trên mạng các máy tính đa lõi”. *Tạp chí Khoa học, Đại học Huế, Tập 121, số 7-A,* 2016.

[65] David B. Kirk, Wen-mei W. Hwu (2017). “Programming Massively Parallel Processors: A Hands-on Approach (Third Edition)”, *Elsevier Inc*. 2017, pp 12-13

[66] K. Li, J. F. Naughton, and J. S. Plank (1994). “Low-latency, concurrent checkpointing for parallel programs”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, Aug. 1994.

[67] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny (1997). “Checkpoint and migration of UNIX processes in the Condor distributed processing system,” *Tech. Rep. UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department*, April 1997.

[68] © Intel Corporation. “Intel® Core™ i3-4330 Processor.”, Available: *https://ark.intel.com/content/www/us/en/ark/products/77769/intel-core-i34330-processor-4m-cache-3-50-ghz.html* (last accessed 10th Sep 2019).

[69] Deborah T. Marr; Frank Binns; David L. Hill; Glenn Hinton; David A. Koufaty; J. Alan Miller; Michael Upton (2002). "Hyper-Threading Technology Architecture and Microarchitecture", *Intel Technology Journal* Q1, 2002. Available: *https://web.archive.org/web/20150217050949/https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/* (last accessed 16th Oct 2020).

[70] J.R. Cordy “2006”. “The TXL Source Transformation Language”, *Science of Computer Programming 61,3* (August 2006), pp. 190-210.

[71] Url Download Txl, Available: *http://www.txl.ca/txl-download.html* (last accessed 10th Nov 2019).

[72] TfiR channel (2014). “Linus Torvalds says GPL v3 violates everything that GPLv2 stood for”, *Debconf 2014*, Available: *https://youtu.be/PaKIZ7gJlRU* (last accessed 26th Oct 2020).

[73] Daniel P. Bovet, Marco Cesati (2005). “Understanding the Linux Kernel”, *Third Edition 3rd Edition, O’Reilly Media*, ISBN: 0-596-00565-2, p3.

[74] Ubuntu manuals : bionic (3) pthread_create.3.gz Available: *http://manpages.ubuntu.com/manpages/bionic/man3/pthread_create.3.html* (last accessed 17th Oct 2019).

[75] Intel Corporation. “Siêu phân luồng là gì?” Tài liệu online tại địa chỉ , *https://www.intel.vn/content/www/vn/vi/gaming/resources/hyper-threading.html*. (Truy cập 05/01/2023).

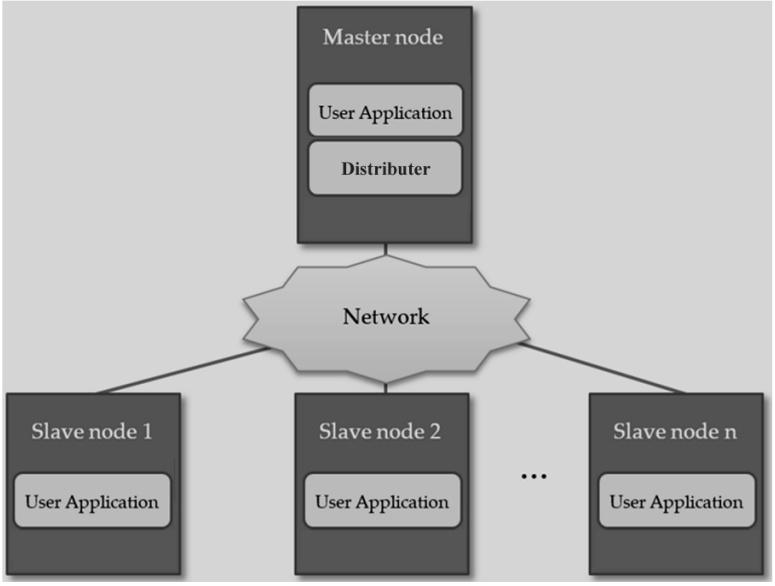
PHỤ LỤC 1: BỘ MÃ NGUỒN, CÁCH THỨC CÀI ĐẶT VÀ SỬ DỤNG THƯ VIỆN CAPE ĐA LUỒNG

Nhóm nghiên cứu đã quyết định sẽ mở mã nguồn của CAPE cho cộng đồng. Do đó, toàn bộ mã nguồn cùng với hướng dẫn sử dụng đã được upload tại: https://github.com/doxuanhuyen

Hiện tại, CAPE được phát triển bằng ngôn ngữ C, trên hệ điều hành Linux Ubuntu Ubuntu 18.04 – Kernel 5.0.0.29.x. Các máy tính làm nút được kết nối mạng cục bộ với nhau.

Biểu đồ triển khai

Cho đến hiện tại, các phiên bản CAPE đều được cài đặt và thử nghiệm trên các cluster gồm các máy tính cá nhân, được nối mạng cục bộ.



Hình 4.5. Biểu đồ triển khai CAPE đa luồng

Hình 4.5 trình bày mô hình triển khai CAPE đa luồng trên 1 cluster bao gồm 1 nút chính và n nút phụ.

Các thành phần của thư viện CAPE

Thư viện CAPE bao gồm các thư mục/file dưới đây:

1) Thư mục src : chứa mã nguồn của thư viện CAPE và các chương trình đã cài đặt sử dụng thư viện CAPE.

+ /src/monitor/cape.c : mã nguồn thư viện CAPE.

+ src/apps : chứa mã nguồn các chương trình viết theo CAPE. Ví dụ như cape_parallel_for_multthread.c là chương trình CAPE đã cài đặt chỉ dẫn paraller_for của OpenMP chạy nhiều tiến trình trên mỗi máy ; cape_reduction_multithread.c là chương trình CAPE đã cài đặt mệnh đề `reduction` của OpenMP chạy nhiều tiến trình trên mỗi máy ;…

2) Thư mục include : chưa file.h của thư viện CAPE và thư viên liên quan (nếu có)

+ /include/cape.h : chưa các định nghĩa biến và tên hàm gọi của thư viên CAPE.

3) Thư mục lib : chứa các file thư viện của CAPE đã biên dịch và thư viện liên quan (nếu có)

+ /lib/cape.o : file thư viện CAPE đã biên dịch.

4) Thư mục bin : chứa các file chương trình CAPE đã biên dịch ra mã máy để chạy.

5) Thư mục / (gốc) : chứa chương trình phân phối, file hướng dẫn cài đặt, file để biên dịch toàn bộ chương trình.

+ /makefile : file script để liên kết và biên dịch chương trình theo mô hình CAPE.

+ /README.md : file hướng dẫn cài đặt và sử dụng CAPE.

+ /ip_config.sh : chứa ip của các máy tính làm nút khi chạy chương trình CAPE.

+ /deploy_cape.sh : file script để copy thư viện và chương trình CAPE đến tất cả các nút theo khai báo thông tin từ file /ip_config.sh.

+ /cape_test.sh : file script chức năng thiết lập hệ thống các nút tham gia vào hệ thống và khởi chạy hệ thống.

Distributer – Trình phân phối

Distributer là chương trình có chức năng thiết lập hệ thống các nút tham gia vào hệ thống và khởi chạy hệ thống.

Distributor được chứa trong file /cape_test.sh

Chương trình này phải được biên soạn cho từng cấu hình hệ thống vận hành CAPE, với các bài toán khác nhau. Trong tương lai chúng tôi có thể viết một chương

trình để tự động sinh ra chương trình này, chỉ yêu cầu người sử dụng nhập vào các thông số hệ thống.

Ví dụ

Ví dụ dưới đây được viết cho 1 hệ thống có 1 nút chính và 2 nút phụ, trong nút chính có địa chỉ IP là 192.168.122.1 và hai nút phụ có IP lần lượt là 192.168.122.179, 192.168.122.223. Các chương trình của CAPE và trình ứng dụng (mulmat) đều được đặt trong thư mục /home/user/cape2/cdv9

Mã của trình phân phối là một trình Shell script, chỉ cho chạy một lần trình ứng dụng trên một nút tính toán như dưới đây (trong đó các số thứ tự được thêm vào để tiện trình bày, trong chương trình thực thi không có).

cape_test.sh

```
1. #!/usr/bin/env bash
2. folder=home/user/cape2
3. prog=cape_parallel_for_multthread4c_1200
4. master=192.168.122.1
5. node1=192.168.122.179
6. node2=192.168.122.223
7. mpirun --host
${master},${node1},${node2}~/${folder}/bin/${prog}
8. echo The execution of $prog is finished.
```

Giải thích các câu lệnh

Số	Ý nghĩa
1	Chỉ định trình thông dịch Shell sh sẽ được sử dụng để chạy chương
2	trình
3	Chỉ định nơi lưu trữ các chương trình của CAPE
4	Chỉ định chương trình ứng dụng được chạy
5-6	IP của nút chính
7	IP của các nút phụ
8	Chạy chương trình CAPE
	Thông báo lên màn hình chương trình đã chạy xong

Với các dòng lệnh trên, hệ thống sẽ được khởi tạo khi người sử dụng cho thực hiện chương trình shell này tại nút chính. Khi đó, từ lệnh 1 đến lệnh 6 để khai báo các

thông số để chạy chương trình gồm tên chương trình sẽ chạy, địa chỉ IP của nút chính và các nút phụ. Lệnh 7 sẽ gọi để chạy chương trình CAPE trên nút chính và 2 nút phụ. CAPE sử dụng thư viện của MPI để gửi/nhận ảnh chụp tiến trình giữa các máy với nhau.

User Application – Trình ứng dụng

Là chương trình ứng dụng của người dùng, có dạng chương trình CAPE đa luồng, được biên dịch từ chương trình OpenMP ban đầu được viết bằng C.

Dưới đây là ví dụ của trình ứng dụng nhân ma trận dạng mã nguồn của CAPE đa luồng. Để được chạy, chương trình này cần được dịch ra mã máy bằng trình duyệt C bình thường có hỗ trợ OpenMP.

```
#include <math.h>
#include <stdio.h>
#include "mpi.h"
#include <sys/time.h>
#include <omp.h>

#include "../../include/cape.h"
void * __get_pc () { return __builtin_return_address(0); }

#define N 2400
int A[N][N], B[N][N], C[N][N];

int i, j,k, THREADS=2;
long sum;
double tm0 = 0.0, tm4 = 0.0 ,tm3=0.0,tm2=0.0,tm1=0.0;

void main(){
 double tm0 = omp_get_wtime();
 cape_declare_variable(&A, CAPE_INT, N*N, 0);
 cape_declare_variable(&B, CAPE_INT, N*N, 0);
 cape_declare_variable(&C, CAPE_INT, N*N, 0);
 cape_declare_variable(&i, CAPE_INT, 1, 0);
 cape_declare_variable(&j, CAPE_INT, 1, 0);
 cape_declare_variable(&k, CAPE_INT, 1, 0);
 cape_declare_variable(&sum, CAPE_LONG, 1, 0);
 cape_init();
 tm0 = omp_get_wtime();
 //load data
 for(i=0;i<N;i++)
       for(j=0;j<N;j++){
                   A[i][j]= 1;
                   B[i][j]= 1;
             }
 tm1 = omp_get_wtime();
 cape_begin(PARALLEL_FOR, 0, N);
 ckpt_start();
 tm2 = omp_get_wtime();
 #pragma omp parallel for schedule(static) private(i,j,k,sum)
shared(C) num_threads(THREADS)
```

```
      for(i = __left__; i < __right__; i ++){
            for(j = 0; j < N; j++ ){
                        sum = 0 ;
                        for ( k = 0; k < N; k++){
                              sum = sum + A[i][k] * B[k][j];
                        }
                        C[i][j] = sum;
            }
      }
      tm3 = omp_get_wtime();
      __pc__ = (unsigned long) __get_pc();
      cape_end(PARALLEL_FOR, FALSE); //TRUE: if exist reduction
clause, FALSE: if it does not
         cape_finalize();
         tm4 = omp_get_wtime();

         printf("%c\t%d\t%d\t%d\t%g\t%g\t%g\t%g\t%g\t
Node:N:Core;TotalTime;initcape;initdata;cal;final\n",
's',THREADS,cape_get_node_num(),N, tm4-tm0,tm1-tm0,tm2-tm1,tm3-
tm2,tm4-tm3);
     }
```

Trong đoạn mã ở trên hàm `cape_begin()` dùng để thực hiện việc ghi lại danh sách các biến có thể chia sẻ và các thuộc tính của chúng. Sau khi thực hiện các công việc trong vùng song song, hàm `cape_end ()` thực hiện một tập hợp các hoạt động để đồng bộ hóa các biến được chia sẻ giữa các nút và làm sạch dữ liệu không cần thiết của bộ nhớ để chuẩn bị cho việc thực thi chương trình ở đoạn mã tiếp theo.

Có hai bước quan trọng của hoạt động chụp ảnh tiến trình trong CAPE: (1) lưu trữ dữ liệu thời điểm ban đầu cần kiểm soát và (2) phát hiện thay đổi của dữ liệu để đồng bộ nếu có. Hàm `ckpt_start()` để thực hiện bước thứ nhất. Khi hàm này được gọi, sẽ sao chép tất cả dữ liệu của các biến có thuộc tính `shared`, `lastprivate`, `copyin` hoặc `copyprivate` (gọi là `DATA-0`) như đã khai báo của chương trình OpenMP được chuyển tương ứng sang cú pháp CAPE `cape_declare_variable`. Bước 2 để tìm được việc thay đổi dữ liệu để đồng bộ, CAPE thực hiện việc so sánh dữ liệu `DATA-0` với dữ liệu hiện thời (đọc dữ liệu từ vùng nhớ của chương trình tại thời điểm đồng bộ). Nhưng dữ liệu thay đổi được lưu vào file ảnh chụp tiến trình. Đối với các biến dạng như reduction tổ chức file ảnh chụp tiến trình cũng có đánh dấu để sử tương ứng khi đồng bộ dữ liệu.

Còn lại hai hàm `cape_init()` và `cape_finalize()` dùng để thiết lập môi trường của chương trình khi chạy song song trên nhiều máy tương ứng với việc gọi lại thư viện môi trường của MPI để CAPE có thể gọi hàm gửi nhận file ảnh chụp tiến trình.

Chạy chương trình

Các bước để chạy thử chương trình CAPE như sau:

1. Copy các file của CAPE vào thư mục /home/user/cape2 trên mỗi nút. Có thể sử dụng lệnh `sh ./deploy_cape.sh`

2. Trên nút chính, mở của sổ lệnh, chuyển thư mục hiện thời đến /home/user/cape2 và chạy câu lệnh sau:

```
sh ./cape_test.sh
```

3. Chờ cho hệ thống thực hiện chương trình và nhận kết quả trên nút chính.

PHỤ LỤC 2: KẾT QUẢ SỐ LIỆU CHI TIẾT THỰC NGHIỆM SO SÁNH HIỆU NĂNG CỦA CAPE ĐA LUỒNG VỚI CAPE ĐƠN LUỒN VÀ MPI

Bảng 4.4. So sánh hiệu năng của CAPE đã luồng với CAPE đơn luồng và MPI

Num of nodes	**Size**	**Core num**	**CAPE (s)**	**MPI (s)**	**OpenMP (s)**	**% MPI/ CAPE**	**OpenMP / CAPE**
a	b	c	d	e	f	g=(1-d/e)*100	h=f/d
16	9600	1	2776	2496	32556	-11%	11.73
16	9600	2	1695	1397	17401	-21%	10.27
16	6400	1	710	644	9292	-10%	13.09
16	6400	2	458	396	4919	-16%	10.74
Average						**-15%**	**11.46**
8	9600	1	4791	4501	34991	-6%	7.30
8	9600	2	2454	2368	17942	-4%	7.31
8	9600	3	1890	1698	13011	-11%	6.88
8	9600	4	1514	1366	10243	-11%	6.77
8	6400	1	993	955	7147	-4%	7.20
8	6400	2	591	589	4488	0%	7.59
8	6400	3	441	440	3386	0%	7.68
8	6400	4	374	365	2653	-2%	7.09
Average						**-5%**	**7.23**