입문자를 위한

병렬 프로그래밍

An Introduction to Parallel Programming

An Introduction to Parallel Programming
by Peter Pacheco

입문자를 위한
병렬 프로그래밍

초판 1쇄 발행 2015년 2월 26일

지은이 피터 파체코
옮긴이 김성민
펴낸이 장성두
펴낸곳 제이펍

출판신고 2009년 11월 10일 제406-2009-000087호
주소 경기도 파주시 문발로 141 뮤즈빌딩 403호
전화 070-8201-9010 / **팩스** 02-6280-0405
홈페이지 www.jpub.kr / **이메일** jeipub@gmail.com

편집부 이민숙, 이 슬, 이주원 / **소통·기획팀** 현지환
본문디자인 디자인콤마
용지 에스에이치페이퍼 / **인쇄** 해외정판사 / **제본** 광우제책사

ISBN 979-11-85890-15-9 (93000)
값 30,000원

입문자를 위한
병렬 프로그래밍
An Introduction to Parallel Programming

피터 파체코 지음
김성민 옮김

Jpub 제이펍

※ 드리는 말씀

- 이 책에 기재한 회사명 및 제품명은 각 회사의 상표 및 등록명입니다.
- 이 책에서는 ™, ©, ® 등의 기호를 생략하고 있습니다.
- 이 책에서 사용하고 있는 실제 제품 버전은 독자의 학습 시점에 따라 책의 버전과 다를 수 있습니다.
- 책의 내용과 관련된 문의사항은 역자나 출판사로 연락해 주시기 바랍니다.
  - 역　자: sungmin.kim0731@gmail.com
  - 출판사: jeipub@gmail.com

차례

CONTENTS

CHAPTER 4 Pthreads를 이용한 공유 메모리 프로그래밍

옮긴이 머리말

TRANSLATOR'S PREFACE

오늘날 병렬 하드웨어(parallel hardware)는 개인용 컴퓨터뿐만 아니라 독자 여러분의 손에 있는 스마트폰에 이르기까지 다양한 곳에서 사용되고 있다. 이렇게 병렬 하드웨어는 대중화되었으나 병렬 하드웨어를 제대로 프로그래밍할 줄 하는 개발자는 아직까지 그리 많지 않다.

이 책은 프로그래밍을 공부하는 대학생(사실 누구라도 상관없다)들이 병렬의 개념부터 병렬 하드웨어, 그리고 병렬 하드웨어를 활용한 병렬 소프트웨어를 학습하도록 설명하고 있다. 1장에서는 병렬의 개념에 대해서 설명하고 있으며, 2장에서는 병렬 하드웨어에 대한 개념을 설명하고 있다. 이 2장의 병렬 하드웨어 개념은 아키텍처에 대한 기본적인 내용부터 병렬 하드웨어까지 일목요연하게 설명하고 있다. 개인적으로 이 2장은 독자 여러분이 꼭 읽어 보기 바란다. 3장부터는 MPI, Pthreads, 그리고 OpenMP를 사용하여 병렬 프로그램을 작성하는 방법을 소개하고 있다. 이 세 개의 기술은 병렬 프로그래밍을 할 때 가장 많이 사용되는 애플리케이션 프로그래밍 인터페이스(API)이다. MPI나 Pthreads, 그리고 OpenMP를 각각 설명하는 책은 제법 있지만, 이렇게 한 권의 책에서 설명하고 있는 것은 지금껏 보지 못했다.

이 책은 미국의 대학에서 컴퓨터 공학과의 학부 교재로도 사용하고 있다. 병렬 프로그래밍을 공부하는 학생들과 병렬 프로그래밍을 실무로 직접 하고 있는 여러 개발자들에게 이 책이 병렬 프로그래밍의 개념부터 응용까지 폭넓은 경험을 줄 것이라 믿어 의심치 않는다.

이 책의 번역을 의뢰받은 것은 꽤 오래 전인데 첫 페이지부터 마지막 페이지까지 한 호흡으로 집필을 한 "피터 파체코(Peter Pacheco)" 교수의 내공을 부족한 역자의 머리로 풀어내기가 쉽지

않았다.

이 책이 나오기까지 우여곡절을 겪고 부족한 저와 함께 머리를 싸매고 고민을 해 준 제이펍 출판사 관계자 분들과 장성두 대표님께 가장 큰 감사를 드린다. 원고가 늦어지는 역자의 게으름을 인내해 주셨기에 이 훌륭한 책이 독자 여러분의 손에 쥐어질 수 있었다. 깊은 내공이 담긴 원서를 번역하면서 지치고 힘들 때마다 응원과 격려를 아끼지 않은 이연구 소장님과 이희규 부장님, 한명구 부장님, 그리고 신인묵 교수님과 박인서 교수님의 멘토링이 없었다면 이 책은 아직도 세상에 나오지 못했거나, 나왔더라도 완성도가 떨어지는 조숙아의 모습이었을 것이다.

아직 채 여물지 않은 햇과일을 시장에 내어놓은 농부의 심정이지만, 이 책을 통해 단 한 명의 개발자라도 병렬 프로그래밍의 깊은 맛을 느껴 보기를 소망한다.

김성민

머리말

PREFACE

병렬 하드웨어는 지금까지 다양한 곳에 널리 퍼져 왔다. 멀티코어 프로세서를 사용하지 않는 랩톱, 데스크톱, 혹은 서버를 찾기는 어렵다. 베어울프(beowulf) 클러스터는 1990년대 존재했던 고성능 워크스테이션이며 오늘날의 조상이라고 볼 수 있다. 클라우드(cloud) 컴퓨팅은 데스크톱처럼 액세스가 가능한 분산 메모리 시스템이다. 그럼에도 불구하고 대부분의 컴퓨터 과학 전공자들은 병렬 프로그래밍에 대해 경험이 없거나 있다 하더라도 소수이다. 많은 학교에서 병렬 컴퓨팅에 대해 상위 몇몇 학생들을 위한 수업을 개설하지만, 대부분의 컴퓨터 과학 전공 학과는 이수할 교육 과정이 너무 많기 때문에 많은 졸업생들이 멀티스레딩이나 멀티프로세스 프로그램에 대해 작성해 본 경험이 거의 없다.

이러한 현상은 분명히 바뀌어야 할 필요가 있다. 많은 프로그램이 하나의 코어에서 만족하는 성능을 얻고 있다고 하더라도 컴퓨터 과학자들은 병렬화를 사용하여 얻을 수 있는 큰 성능 향상에 대해 눈여겨봐야 한다. 그리고 필요할 때 이러한 병렬화의 잠재력을 사용할 수 있어야 한다.

이 책은 이러한 문제를 어느 정도 해결하기 위해 집필됐다. 이 책은 MPI, Pthreads, 그리고 OpenMP를 사용하여 병렬 프로그램을 작성하는 방법을 소개하고 있다. 이 세 개의 기술은 병렬 프로그래밍을 할 때 가장 많이 사용되는 애플리케이션 프로그래밍 인터페이스(API, Application Program Interface)이다. 필자가 생각한 독자는 병렬 프로그램을 작성해야 하는 학생들과 그 방면의 종사자들이다. 이 책을 읽기 전에 알고 있어야 할 최소한의 지식은 대학에서 배운 수학과 C를 사용하여 시리얼(serial) 프로그램을 작성할 수 있는 능력 정도이다. 이것

은 학생들이 가능한 한 일찍 병렬 시스템을 프로그래밍해야 할 때 필요한 최소한의 능력이라고 생각한다.

샌프란시스코 대학(University of San Francisco)에서 컴퓨터 과학 전공 학생들은 대부분의 1학년생이 수강해야 하는 "Introduction to Computer Science I" 과목을 이수한 후에 이 책을 사용하는 수업을 의무적으로 수강해야 한다. 지난 6년 동안 병렬 컴퓨팅에서 이 과목을 개설해 왔고 학생들이 고학년이 될 때까지 병렬 프로그램을 작성하는 데 주저하지 않도록 많은 경험을 익히게끔 해 왔다. 이 수업은 인기가 많으며 학생들이 "Introduction" 수업을 수강한 후에 다른 수업과 이 수업을 병행하는 것이 더 쉽다는 것을 알게 됐다.

2학년이 병렬 프로그램을 작성하는 방법에 대해 배운다면 교수들은 학생들 스스로가 병렬 프로그램 작성 방법을 익히도록 해 줘야 한다. 이 책이 그들에게 유용한 자원이 될 것이라 믿어 의심치 않는다.

이 책에 대하여

앞에서 말했지만, 이 책의 주된 목적은 MPI, Pthreads, 그리고 OpenMP를 사용하여 컴퓨터 과학에 대한 배경 지식이 부족한 학생들이나 병렬화에 대해 경험이 없는 학생들에게 병렬 프로그래밍을 가르치는 것이 목적이다. 가능한 한 쉽고 유연하게 설명하여 한두 개의 API에 대해서 별다른 흥미를 갖지 못하는 독자들이 적은 노력으로 나머지 내용들을 모두 익힐 수 있도록 해 준다. 따라서 세 API에 대한 장은 서로 독립적이다. 어떤 순서로 읽어도 상관없으며 그중 한두 개는 그냥 넘어가도 된다. 이러한 독립적인 내용 구성이 이 책의 장점이다. 이 세 장에서 필요한 내용은 반복해서 설명했다. 물론 반복된 내용은 간단하게 짚고 넘어가거나 그냥 넘겨도 상관없다.

병렬 컴퓨팅에 경험이 없는 독자들은 먼저 1장을 읽기 바란다. 병렬 컴퓨팅이 컴퓨터 분야에서 중요한 역할을 하는 이유에 대해 상대적으로 덜 기술적으로 설명하고 있다. 1장은 또한 병렬 시스템과 병렬 프로그래밍에 대해 간단하게 소개하고 있다. 2장은 컴퓨터 하드웨어와 소프트웨어에 대한 기술적 배경 지식을 제공한다. 하드웨어에 대한 대부분의 내용은 API장을 읽기 전에 간략하게나마 읽어 보기 바란다. 3장, 4장, 그리고 5장은 MPI, Pthreads, 그리고 OpenMP를 이용한 프로그래밍을 소개하고 있는 장이다.

6장에서는 두 개의 큰 프로그램을 개발한다. 하나는 병렬 n-body 솔루션이고 다른 하나는 병렬 트리 검색이다. 이 두 프로그램은 세 개의 API를 모두 사용해서 개발한다. 7장은 다양한

측면의 병렬 컴퓨팅에 대해 추가적인 정보를 간략하게 다루고 있다.

C 언어를 사용하여 프로그램을 개발하는데, 그 이유는 이 세 가지 API가 모두 C 언어 인터페이스를 갖고 있으며 C 언어는 사용하기에 상당히 간단한 언어이고 배우기도 상대적으로 쉬운 언어이기 때문이다. 특별히 C++이나 자바 개발자들은 이미 C 언어의 제어 구조에 대해 익숙하리라 생각한다.

수업에서의 사용

이 책은 샌프란시스코 대학의 학부 저학년을 대상으로 했다. 이 책을 교재로 사용하는 수업은 컴퓨터 전공자들에게 학부 운영 체제 수업을 수강하기 위한 필수 과목이다. 이 과정을 듣기 위해 필요한 것은 한 학기의 "Introduction to computer science" 과목에서 "B학점" 이상을 받거나 두 학기의 "Introduction to computer science"에서 "C학점" 이상을 받으면 된다. 수업은 처음 4주는 C 프로그래밍에 대한 소개로 시작한다. 대부분의 학생들이 자바 프로그램에 익숙하기 때문에 C 프로그래밍에 대한 수업은 주로 C 언어에서의 포인터 사용을 설명하고 있다.[1] 1장, 3장, 4장, 그리고 5장의 대부분을 강의하며 2장과 6장은 그중에서 중요한 부분만 설명한다. 2장에 대한 배경 지식은 필요한 경우에만 소개한다. 예를 들어, OpenMP9(5장)에서 캐시 일관성(cache coherence) 문제를 설명하기 전에 2장에서 캐시에 대한 내용을 설명한다.

수업은 매주 숙제가 있으며 다섯 개의 프로그래밍 숙제, 두 번의 중간고사, 그리고 한 번의 기말고사로 구성되어 있다.

숙제는 대부분 짧은 프로그램을 작성하거나 기존의 프로그램을 약간 수정하는 내용이다. 이렇게 하는 목적은 학생들이 현재 수업을 잘 따라오도록 하고 수업 시간에 소개한 개념에 대해 직접 경험해 보라는 것이 그 이유이다. 숙제를 해야 하는 것은 그 수업을 성공적으로 끝내기 위한 중요한 이유 중의 하나이다. 이 책에 있는 대부분의 연습문제는 이러한 간단한 숙제로 사용하기에 적합하도록 구성되어 있다. 프로그래밍 숙제는 일반적인 숙제로 작성하는 프로그램보다는 좀 더 큰 규모이다. 그러나 학생들에게는 좋은 경험이 될 것이다. 종종 숙제에 의사코드(pseudocode)를 포함하거나 수업 시간에 좀 더 어려운 내용을 다루곤 한다. 이 추가적인 내용이 중요하다. 학생들이 프로그램을 완성할 때까지 너무 긴 시간이 걸리는 숙제를 내주는 것은 어려운 일이 아니다. 중간고사와 기말고사의 결과, 그리고 운영 체제를 가르치는 교

1 흥미롭게도 많은 학생들이 C 포인터를 사용하는 것이 MPI 프로그래밍보다 더 어렵다고 말한다.

수들의 열정적인 리포트는 이 수업이 실제로 학생들에게 병렬 프로그램을 작성하는 방법을 가르치는 데 매우 성공적이었다는 것을 말해 준다.

병렬 컴퓨팅의 더 고급 수업을 위해서 이 책과 온라인 지원이 제공된다. 제공되는 것들은 이 책에서 설명하지 못한 세 개의 API에 대한 문법(syntax)과 그 외 많은 정보들이다. 이 책은 또한 프로젝트 기반 수업이나 병렬 연산을 이용하는 컴퓨터 과학 이외의 분야에서도 수업에 사용할 수 있다.

지원

책의 웹사이트는 http://www.mkp.com/pacheco이다. 정오표와 관련된 자료 사이트에 대한 링크를 포함하고 있다. 교수들은 완전한 강의 노트, 책에 있는 그림, 그리고 연습문제와 프로그래밍 문제에 대한 답을 다운로드할 수 있다. 모든 사용자들은 책에서 다룬 긴 프로그램을 다운로드할 수 있다.

여러분들이 오류를 찾아 연락을 주면 매우 고맙게 생각할 것이다. 혹시 필자의 실수를 찾게 되면 peter@usfca.edu로 이메일을 보내 주기 바란다.

감사의 글

ACKNOWLEDGMENTS

이 책으로 수업하면서 많은 사람들에게 도움을 받았다. 많은 사람들 중에서 초기 제안서를 읽어 주고 조언을 해 준 다음의 분들께 감사드린다. 피크렛 어칼(Fikret Ercal)[미주리 과학기술대학(Missouri University of Science and Technology)]과 댄 하비(Dan Harvey)[서던오리건 대학(Southern Oregon University)], 조엘 홀링스워스(Joel Hollingsworth)[엘론 대학(Elon University)], 젠스 마셰(Jens Mache)[루이스앤클락 대학(Lewis and Clark College)], 돈 맥러플린(Don McLaughlin)[웨스트버지니아 대학(West Virginia University)], 매니시 파라샤(Manish Parashar)[럿거스 대학(Rutgers University)], 찰리 펙(Charlie Peck)[얼햄 대학(Earlham College)], 스티븐 렝크(Stephen C. Renk)[노스센트럴 대학(North Central College)], 롤프 조지프 사센펠드(Rolfe Josef Sassenfeld)[텍사스 대학 엘파소 캠퍼스(The University of Texas at El Paso)], 조지프 슬론(Joseph Sloan)[워포드 대학(Wofford College)], 미셸라 토퍼(Michela Taufer)[델라웨어 대학(University of Delaware)], 펄 왕(Pearl Wang)[조지메이슨 대학(George Mason University)], 밥 윔즈(Bob Weems)[텍사스 대학 알링턴 캠퍼스(University of Texas at Arlington)], 그리고 수청종(Cheng–Zhong Xu)[웨인 주립대학(Wayne State University)].

또한, 이 책의 여러 장을 읽고 검토를 해 준 다음의 분들께도 깊은 감사를 드린다. 던컨 뷰엘(Duncan Buell)[사우스캐롤라이나 대학(University of South Carolina)], 머사이어스 고버트(Matthias Gobbert)[메릴랜드 볼티모어 카운티 주립대학(University of Maryland, Baltimore County)], 크리슈나 카비(Krishna Kavi)[노스텍사스 대학(University of North Texas)], 린홍(Hong Lin)[휴스턴–다운타운 대학(University of Houston–Downtown)], 캐시 리즈카(Kathy Liszka)[애

크런 대학(University of Akron)], 리 리틀(Leigh Little)[뉴욕 주립대학(The State University of New York)], 리우신리안(Xinlian Liu)[후드 대학(Hood College)], 헨리 투포(Henry Tufo)[콜로라도 대학 볼더 캠퍼스(University of Colorado at Boulder)], 앤드루 슬로스Andrew Sloss [ARM 컨설턴트 엔지니어(Consultant Engineer, ARM)], 그리고 정겅빈(Gengbin Zheng)[일리노이 대학(University of Illinois)]께 감사를 드린다. 이 분들의 조언과 제안은 이 책을 더욱 값지게 해 주었다. 물론 아직까지 책에 남아 있는 오류나 실수는 모두 저자의 책임이다.

캐시 리즈카(Kathy Liszka)는 이 책을 사용하는 교수들을 위해 슬라이드를 준비해 줬고, 졸업생인 최진영(Jinyoung Choi)은 솔루션 매뉴얼을 만들어 줬다. 이 두 명에게 감사를 전한다.

모건 카프만(Morgan Kaufmann)의 직원들에게 이 프로젝트 전반에 걸쳐 매우 고맙게 생각한다. 특별히 개발 에디터인 네이트 맥패든(Nate McFadden)에게 깊은 감사를 전한다. 그는 매우 가치 있는 충고를 주었고 리뷰를 위한 귀찮은 할당 작업을 맡아 주었다. 또한 지난 몇 년 동안 발견되는 이 책의 문제점들에 대해 인내심을 갖고 지켜봐 주었다. 매릴린 래시(Marilyn Rash)와 메건 기니(Megan Guiney)에게도 감사를 전한다. 이 두 사람은 책이 출판되기까지 일을 효율적으로 해 줬다.

샌프란시스코 대학(USF)의 컴퓨터 과학 분야와 수학 분야에 종사하는 나의 동료들이 이 책을 집필하는 동안 많은 도움을 줬다. 특별히 조지 벤슨(Gregory Benson) 교수에게 감사를 드린다. 그의 병렬 컴퓨팅, 그중에서 특별히 Pthreads와 세마포어에 대한 이해는 나에게 가치를 따질 수 없는 큰 자원이 됐다. 또한 시스템 관리자인 알렉세이 페도소프(Alexey Fedosov)와 콜린 빈(Colin Bean)에게도 감사를 드린다. 이 책의 프로그램을 작성할 때 "위급 상황"에서도 끈기 있고 효과적으로 대처해 줬다.

마지막으로 나의 친구 홀리 콘(Holly Cohn), 존 딘(John Dean)과 로버트 밀러(Robert Miller)의 격려와 정신적 지원에 대해 감사를 드리지 않을 수가 없다. 저자의 매우 어려운 시간 동안에 많은 도움을 줬다. 그들에게 영원히 감사한다. 나는 나의 학생들에게 가장 큰 빚을 지고 있다. 그들은 어떤 부분은 너무 쉽고, 어떤 부분은 너무 어려운지를 알려 줬다. 간략하게 말하면 나의 학생들은 나에게 병렬 컴퓨팅을 가르치는 방법에 대해 알게 해 줬다. 이 모든 분들에게 나의 깊은 감사를 전한다.

베타리더 후기

BETA READERS' EPILOGUE

이재빈(연세대학교)

병렬 프로그래밍에 관심이 있어도 관련 참고문헌이 많지 않아 학습하는 데 많은 어려움을 겪은 사람들이 꽤 많을 것으로 생각됩니다. 이제는 그럴 필요 없이 이 책 하나로 병렬 프로그래밍의 기초를 잡고 병렬 프로그래밍이 가져다주는 쾌속 처리의 희열을 함께 느낄 수 있으면 좋겠습니다.

강미희(휴인시스템)

CPU 코어의 수는 계속 늘어나고 있지만, 늘어난 여러 개의 코어를 프로그래밍적으로 어떻게 활용할지에 대해서는 고민해 본 적이 없었습니다. 하지만 이번 베타리딩은 확장된 코어를 제대로 사용할 수 있도록 병렬 프로그래밍 적용 방법을 고민해 볼 좋은 기회였던 것 같습니다.

김정택(포항공과대학교)

병렬 프로그래밍을 처음 접하는 사람들에게 적합한 도서라고 판단됩니다. 병렬 프로그래밍을 시작하기 전에 컴퓨터의 기본 구성 요소인 하드웨어와 소프트웨어 측면에서 이해해야 할 부분이 잘 정리되어 있고, 이를 기초로 분산 및 공유 메모리 프로그래밍에 대해 체계적으로 설명하고 있습니다. 후반부에서는 복잡한 모델에 대한 병렬 프로그래밍 방법을 설명합니다. 이러한 접근은 독자의 이해도를 높여 주며 병렬 프로그래밍을 이해하기 위한 전반적인 지식을 제공한다고 봅니다.

도경원(NHN)

요즘 싱글코어 시스템은 흔치 않습니다. 주변의 PC만 보아도 멀티코어를 사용합니다. 이 책은 멀티코어 시스템을 보다 효율적으로 사용할 수 있는 병렬 프로그래밍 기법인 MPI, Pthreads, OpenMP에 대하여 필요한 내용을 잘 다루고 있습니다. 각 기법은 하나의 장으로 구성되어 필수적인 내용과 병렬 프로그래밍 시에 발생할 수 있는 문제들을 다루고 있습니다.

서지연(다음카카오)

"병렬 프로그래밍이란 무엇일까?"라는 질문으로 베타리딩을 시작했는데, 어느새 책에 흠뻑 빠져서 병렬 프로그래밍적으로 사고하고, 내가 진행 중인 프로젝트에 어떤 식으로 적용해 볼 수 있을까라는 생각으로 머릿속이 가득 찼습니다. 책을 통해 MPI, Pthreads, OpenMP라는 키워드를 얻었고, 이제 그 바탕을 통해 병렬 프로그래밍에 대해 더 깊이 공부해 나갈 수 있을 것 같습니다.

전상우(삼성전자소프트웨어멤버십)

병렬 프로그래밍에 대한 아무런 사전 지식 없이 막연한 '호기심'에 이끌려 책을 읽었습니다. 읽는 과정이 쉽지만은 않았지만, 친절한 설명과 예제 덕분에 병렬 프로그래밍에 대해 제대로 이해할 수 있었습니다. C 언어가 익숙하고 병렬 프로그래밍에 대한 호기심과 학습 열정만 있다면 충분히 도전할 만한 가치가 있는 책이라고 생각합니다.

1

CHAPTER

왜 병렬 컴퓨팅인가?

1986년부터 2002년까지 마이크로프로세서의 성능 증가는 매년 평균 50%였다[27]. 이 믿기 어려운 증가 속도가 의미하는 것은 사용자와 소프트웨어 개발자들이 애플리케이션의 성능을 증가시키기 위해서는 현재 애플리케이션의 최적화를 할 것이 아니라, 그저 다음 세대의 마이크로프로세서가 출현하기를 기다리기만 하면 된다는 것을 의미한다. 그러나 2002년부터 싱글 프로세서의 성능 증가는 1년에 약 20% 정도에 그칠 만큼 느려졌다. 이 차이는 극적이다. 매년 50%라는 성능 증가는 10년 동안 거의 60배의 향상을 이루었지만, 20%의 성능 증가는 단지 6배 향상에 그쳤다.

더 나아가, 성능 증가의 이러한 차이는 프로세서 설계의 극적인 변화와도 관련이 있다. 2005년에 마이크로프로세서의 주요 제조사 대부분은 빠른 성능 증가를 위해 병렬화를 채택하기로 결정했다. 모놀리식(monolithic) 프로세서에서 더 빠르게 하기보다는 하나의 통합된 기판 위에 여러 개의 완벽한 프로세서를 배치하기 시작했다.

이러한 변화는 소프트웨어 개발자들에게도 매우 중요한 결과를 가져왔다. 간단히 말해서, 더 많은 프로세서를 추가하는 것이 싱글 프로세서에서 동작하도록 만들어진 시리얼(serial) 프로그램의 성능을 환상적으로 향상시키지는 않는다는 점이다. 이러한 프로그램들은 다중 프로세서의 존재에 대해서 거의 모르며, 다중 프로세서가 있는 시스템에서 그런 프로

그램의 성능은 다중 프로세서 시스템의 싱글 프로세서에서 실행하는 성능과 거의 같다.

다음의 질문들에 대해 생각해 보자.

1. 왜 우리가 이런 것들을 고려해야 하는가? 싱글 프로세서 시스템으로도 충분히 빠르지 않는가? 결국, 1년에 20%의 성능 증가만으로도 충분히 의미 있는 것 아닌가?
2. 왜 마이크로프로세서 제조업체들은 더 빠른 싱글 프로세서 시스템을 개발하지 못하는가? 왜 **병렬 시스템**(parallel system)을 구축해야 하는가? 왜 다중 프로세서로 구성된 시스템을 구축해야 하는가?
3. 왜 시리얼 프로그램을 현재 다중 프로세서가 잘 사용될 수 있는 **병렬 프로그램**(parallel program)으로 자동으로 변환해 주는 프로그램을 만들 수 없는가?

위의 질문에 대한 간략한 답을 알아보자. 그 답 중에서 몇 개는 정답이라고 할 수는 없다. 예를 들어, 매년 20%는 수많은 애플리케이션에게는 상당히 높은 수치이다.

1.1 성능 증가의 필요성

지난 십수 년 동안 경험한 컴퓨팅 성능의 폭발적인 증가는 과학, 인터넷, 그리고 엔터테인먼트 등의 다양한 분야에서 수많은 극적인 출현을 이끌어 온 원동력이었다. 예를 들어, 인간 게놈, 더욱 정확한 의료 이미지, 놀랄 만큼 빠르고 정확한 웹 검색, 더욱 현실감 있는 게임은 이러한 컴퓨팅 성능의 증가가 없었다면 불가능했을 것이다. 실제로 근래의 컴퓨팅 성능의 증가는 점점 더 어려워지고 있으며, 불가능하지는 않지만 초창기의 성능 증가에는 못 미치고 있다. 컴퓨팅 성능이 증가할수록 우리가 해결해야만 하는 심각한 문제들도 역시 점점 증가한다. 다음의 예를 보자.

- **기후 모델링**(climate modeling): 기후 변화를 좀 더 잘 이해하기 위해서 더욱 정확한 컴퓨터 모델이 필요하고, 이 모델은 대기, 해양, 지표, 그리고 극지방의 빙하 등의 상호 작용을 포함하고 있다. 이러한 다양한 상호관계가 어떻게 지구 전체의 기후에 영향을 미치는지에 대해 더욱 자세히 연구해야 한다.
- **단백질 폴딩**(protein folding): 잘못 폴딩된(misfolding) 단백질은 헌팅턴(Huntington), 파킨

슨(Parkinson), 알츠하이머(Alzheimer)와 같은 질병에 포함되어 있다고 알려져 있지만, 현재의 컴퓨팅 성능으로는 이러한 단백질과 같은 복잡한 분자의 구조를 연구하기에는 매우 제한적이다.

- **의약품 개발(drug discovery)**: 컴퓨팅 성능이 증가함에 따라 신약을 개발하는 과정에 사용되는 사례가 상당히 많다. 예를 들어, 많은 약들이 여러 질병의 상당히 작은 부분의 치료에 효과적이다. 현재까지 알려진 치료의 효과가 없는 경우에 해당 개인의 게놈(genome) 전체를 분석해서 대체할 수 있는 신약을 개발할 수도 있다. 그러나 이러한 게놈 분석을 위해서는 엄청난 컴퓨팅 성능이 필요하다.
- **에너지 연구(energy research)**: 컴퓨팅 성능이 증가할수록 풍속 터빈, 태양열 전지, 배터리 등의 더욱 구체적인 기술 모델을 연구하는 프로그램 제작이 가능해진다. 이 프로그램은 더욱 효율적인 클린 에너지 소스를 구축하는 데 필요한 정보를 제공할 수 있다.
- **데이터 분석(data analysis)**: 현재 우리는 엄청난 양의 데이터를 생성하고 있다. 전 세계에서 저장하는 데이터의 양은 2년마다 두 배씩 증가한다고 한다[28]. 그러나 이 중 대부분은 분석하지 않으면 쓸모가 없는 형태다. 예를 들어, 인간 DNA 뉴클레오티드(nucleotide)의 배열을 알고 있다면 그 자체만으로는 별로 소용이 없다. 이러한 순서가 개발에 어떻게 영향을 미치고, 질병을 어떻게 일으키는지를 이해하기 위해서는 막대한 분석이 필요하다. 게놈 데이터와 마찬가지로 유럽원자핵공동연구소(CERN, Conseil Européen pour la Recherche Nucléaire)에 있는 강입자충돌기(large hadron collider)와 같은 원자 충돌기에서도 엄청난 양의 데이터가 생성된다. 또한, 의료 영상, 천문학 연구, 웹 검색 엔진 역시 엄청난 데이터를 생성하기는 마찬가지다.

위에서 언급한 연구뿐만 아니라 다른 문제들도 컴퓨터 성능의 증가 없이는 해결할 수 없는 문제들이 많다.

1.2 병렬 시스템을 구축해야 하는 이유

싱글 프로세서 성능의 놀라운 증가는 집적회로의 트랜지스터, 즉 전자회로의 밀집도 증가에 기인한다. 트랜지스터의 크기가 작아지면 작아질수록 트랜지스터의 속도는 증가되며,

집적회로(integrated circuit)의 전체 속도도 증가된다. 그러나 트랜지스터 속도의 증가는 전력 소모도 증가시킨다. 이 전력 대부분은 열로 발산되며, 집적회로가 뜨거워지면 그 회로의 기능에 대한 신뢰성이 떨어진다. 21세기 초반에 공랭식(air-cooled) 집적회로는 공기만으로 열을 식히는 데 한계에 직면했다[26].

그러므로 집적회로의 속도를 계속 증가시키는 것이 점점 불가능해졌다. 그러나 트랜지스터 자체의 집적도는 적어도 당분간은 계속 증가할 수 있다. 또한, 현재의 컴퓨팅 능력을 높이기 위해 주어진 잠재력을 끌어내는 것은 컴퓨팅 능력을 계속 증가시키기 위해서 피할 수 없는 선택이다. 결국 집적회로 산업이 새롭고 더 나은 제품을 계속 개발해 내지 못하면 그 산업은 점차 소멸할 수밖에 없다.

어떻게 트랜지스터 집적도를 계속 증가시켜 나갈 수가 있을까? 정답은 **병렬화(parallelism)**다. 더 빠르고 더 복잡한 모놀리식(monolithic) 프로세서를 구축하기보다는, 상대적으로 간단하고 완벽한 프로세서 여러 개를 하나의 칩에 넣기로 결정했다. 그러한 집적회로를 **멀티코어(multicore)** 프로세서라고 하며, **코어(core)**는 중앙처리장치(central processing unit), 즉 CPU와 동의어다. 하나의 CPU로 구성된 기존의 프로세서를 **싱글-코어(single-core)** 시스템이라고 한다.

1.3 병렬화 프로그래밍이 필요한 이유

대부분의 프로그램은 하나의 코어만을 고려하는 일반적인 방법으로 프로그래밍되기 때문에 멀티코어의 존재에 대해 고려하지 않았다. 멀티코어 시스템에서 하나의 프로그램에 대한 다중 인스턴스(multiple instance)를 생성하여 실행하기도 하지만, 이것은 정확하게 말하면 멀티코어를 고려했다고 말하기는 어렵다. 예를 들어, 우리가 좋아하는 게임 프로그램을 여러 개의 인스턴스를 생성하여 실행하는 것은 실제로 우리가 원하는 것이 아닐 수도 있다. 우리가 원하는 것은 여러 개의 인스턴스로 실행되는 것이 아니라, 좀 더 빠르게 실행되며 보다 화려한 그래픽 등을 보는 것이다. 이러한 일을 하기 위해서는 시리얼 프로그램을 **병렬 프로그램**으로 다시 작성해야 하며, 병렬 프로그램을 재작성해야 멀티코어를 사용할

수가 있다. 혹은 시리얼 프로그램을 병렬 프로그램으로 자동으로 변환해 주는 변환 프로그램을 이용하기도 한다. 나쁜 소식은 C나 C++로 작성된 시리얼 프로그램을 병렬 프로그램으로 변환해 주는 프로그램은 제약이 많다는 점이다.

그렇다고 해서 놀랄 만한 것은 아니다. 시리얼 프로그램에서 공통 부분을 인식하는 프로그램을 작성하듯이 자동으로 이러한 부분을 효율적인 병렬 코드로 변환해 주는 것이며, 병렬 부분을 순차적으로 실행하도록 작성하면 상당히 비효율적인 코드가 된다. 예를 들어, 내적(dot product) 연산으로 두 개의 $n \times n$ 행렬의 곱셈을 생각해 보자. 행렬 곱셈을 순차적인 병렬 내적 연산으로 병렬화하면 상당히 느린 결과를 얻게 된다.

시리얼 프로그램의 효율적인 병렬 구현은 각 단계별로 효율적으로 병렬화한다고 해서 얻어지는 것은 아니다. 오히려 가장 최고의 병렬화는 각 단계 이전에 전체적으로 완전히 새로운 알고리즘을 개발하는 것이 더 최선의 방법일 수도 있다. 예를 들어, n개의 값을 계산하고 그 값을 덧셈하는 경우를 가정해 보자. 다음의 시리얼 코드로 원하는 결과를 얻을 수 있다.

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

이제 p개의 코어가 있고, p는 n보다는 훨씬 작다고 가정해 보자. 각 코어는 n/p의 수만큼 부분 합의 형태로 변경할 수 있다.

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last i; my_i++) {
   my_x = Compute_next_value(. . .);
   my_sum += my_x;
}
```

여기서 my_라고 하는 접두어는 각 코어가 자신의 로컬 변수 혹은 프라이빗 변수(private variable)를 사용하며, 각 코어는 이 코드 블록을 다른 코어와는 상관없이 독립적으로 실행한다는 것을 의미한다.

각 코어가 이 코드의 실행을 완료한 후에 변수 my_sum에는 Compute_next_value를 호출해서 계산한 값의 합이 저장된다. 예를 들어, 코어가 8개이고, $n = 24$이면 24번의 Compute_next_value를 호출하고, 호출 결과는 다음과 같다.

1,4,3, 9,2,8, 5,1,1, 6,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9,

my_sum에 저장된 값은 다음과 같다.

core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

여기서 각 코어는 0, 1,..., $p - 1$까지의 범위 중에서 음수가 아니라고 가정한다. 여기서 p는 코어의 수다.

코어가 my_sum 값의 계산을 완료하면 그 결과를 "마스터" 코어로 설계된 코어에 해당 결과를 전송해서 전체 합을 계산한다. 그 부분에 대한 코드를 추가하면 다음과 같다.

```
if (마스터 코어라면) {
   sum = my_x;
   for 각 코어에서는 {
      코어들로부터 값을 수신함;
      sum += value;
   }
} else {
   my_x를 마스터 코어에 전송함;
}
```

결과를 보면, 마스터 코어를 코어 0이라고 했을 때 그 값은 8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95가 된다.

그러나 이보다 더 좋은 방법이 있다는 것을 눈치챘을 것이다. 특히, 코어 수가 많다면 더욱 그렇다. 마스터 코어에게 최종 합계를 구하는 작업 전부를 맡기는 것보다는 각 코어들을 쌍(pair)으로 묶어서 코어 0은 코어 1과의 합을 계산하고, 코어 2는 코어 3과의 합을 계산하고, 코어 4는 코어 5와의 합을 계산하도록 하는 방법이다. 이러한 작업은 짝수 코어에서 실행하게 된다. 반복해서 코어 0은 코어 2와의 합을 계산하고, 코어 4는 코어 6과의 합을

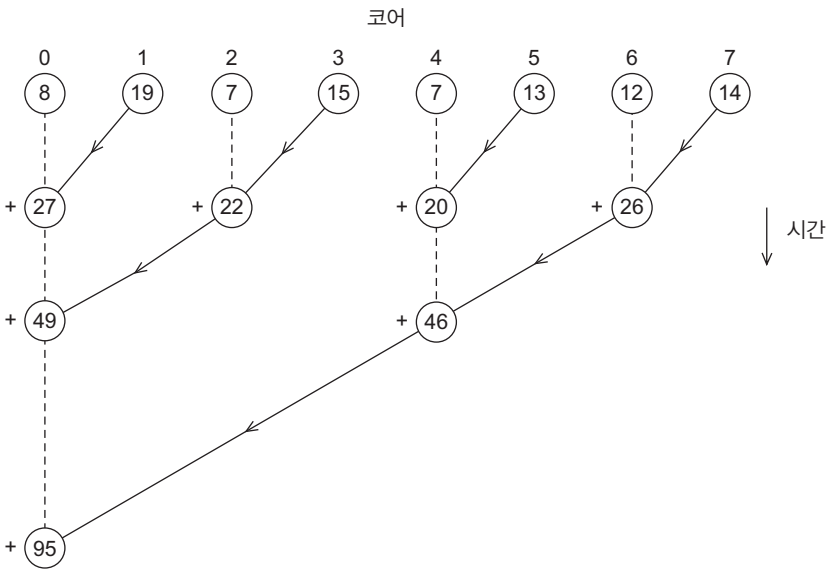

그림 1.1

다중 코어(multiple cores)가 전역 합을 구하는 방식

계산하는 식이다. 이 과정을 네 번 반복할 수 있다. 그림 1.1을 보자. 원(circle)은 각 코어의 현재 합을 나타내며, 화살표 선은 하나의 코어가 다른 코어에 그 합을 보내는 것을 의미한다. 더하기 기호는 코어가 다른 코어로부터 합을 받아서 자신의 합과 덧셈하는 것을 의미한다.

"전역(global)" 합에서 마스터 코어(코어 0)는 다른 코어보다 더 많은 일을 하며, 프로그램이 최종 합을 구하는 데 걸리는 시간은 마스터 코어가 완료하는 시간에 따라 결정된다. 그러나 8개의 코어가 있는 경우, 첫 번째 방법에서 마스터 코어는 다른 코어로부터 총 7번의 합을 수신받아서 덧셈을 하게 되지만, 두 번째 방법은 단지 세 번만 이 작업을 해 주면 된다. 따라서 두 번째 방법의 결과는 두 배 이상의 성능이 향상된다.

이 결과의 차이는 코어의 수가 증가하면 할수록 더욱 커진다. 1,000개의 코어가 있는 경우 첫 번째 방법은 999번을 수신하고 덧셈을 하지만, 두 번째 방법은 단지 10번의 수신만 필요하고, 결국 거의 100배의 성능 증가를 얻을 수 있다.

첫 번째 전역 합은 시리얼 형태의 전역 합을 일반화시킨 평이한 방법이다. 코어별로 덧셈

작업을 나누고, 각 코어는 자신의 합을 계산한 후에 마스터 코어는 단순히 기본적인 시리얼 덧셈을 반복한다. p개의 코어가 있다면 p만큼의 덧셈이 필요하다. 한편, 두 번째 전역 합은 처음의 시리얼 덧셈과 많은 차이가 있다.

여기서 알아야 할 점은 변환 프로그램으로는 두 번째 전역 합 알고리즘을 "만들어 내기"가 쉽지 않다는 점이다. 오히려 효율적인 전역 합을 미리 정의하고 변환 프로그램을 통해 그 정의된 알고리즘의 형태로 변환하는 것이 일반적이다. 원래의 시리얼 루프(loop)를 "인식"하고 그 인식된 부분을 미리 코딩되어 있는 효율적이고 병렬화된 전역 합의 코드로 변환하는 것이다.

시리얼 코드의 많은 부분이 공통 부분으로 인식되고 효율적으로 **병렬화(parallelized)**되도록 수정하여 다중 코어를 사용할 수 있도록 하기 위한 소프트웨어를 개발할 수 있지 않을까 기대할 수도 있다. 그러나 시리얼 프로그램이 복잡하면 할수록 그 내부를 병렬화 가능 부분으로 인식하기는 너무 어렵다. 따라서 그러한 부분을 효율적인 병렬화 코드로 미리 코딩하기는 거의 불가능에 가깝다.

시리얼 프로그램을 작성하지 않고 병렬 프로그램을 작성해야 프로그램이 멀티 프로세서의 능력을 제대로 이용할 수 있다.

1.4 병렬화 프로그램을 작성하는 방법

이 질문에는 몇 가지 답을 제시할 수 있지만, 사실 이 질문에 대한 해답은 코어에서 동작하는 각각의 작업량을 **파티셔닝(partitioning)**한다는 기본적인 개념에 따라 몇 가지로 다르게 나타난다. 크게 보면 **태스크 병렬화(task-parallelism)**와 **데이터 병렬화(data-parallelism)**라는 두 가지 방법이 있다. 태스크 병렬화는 문제를 해결하기 위해 여러 가지 태스크를 각 코어에 파티셔닝, 즉 할당하는 방법이다. 데이터 병렬화는 문제를 해결하기 위해 사용할 데이터를 쪼개어 각 코어에 파티셔닝, 즉 할당하는 방법이다.

예를 들어, P 교수가 "영문학" 강의를 한다고 가정해 보자. P 교수의 수업을 듣는 학생 수는 100명이며, 네 명의 조교 A, B, C, D를 배정했다고 가정해 보자. 학기가 끝날 때쯤 P

교수는 다섯 문제로 구성된 기말고사를 보기로 했다. 이 기말고사를 채점하기 위해서 P 교수와 네 명의 조교들은 다음과 같은 두 가지 선택을 고민하게 됐다. P 교수와 네 명의 조교는 한 문제당 학생 100명의 기말고사를 채점할 수 있다. 예를 들면, P 교수는 1번 문제, A 조교는 2번 문제를 맡아서 채점하는 방법이다. 또 하나의 선택은 한 명당 전체 기말고사 중에 20명 분의 문제지를 채점하는 방법이다. 전체 학생 수 100명을 다섯 개의 파일로 묶어서 P 교수는 첫 번째 학생들의 문제지를 채점하고, A 조교는 그 다음 학생들의 문제지를 채점하는 방법이다.

위의 두 가지 방법에서 교수와 조교들은 "코어"를 의미한다. 첫 번째 방법은 태스크 병렬화의 예라고 볼 수 있다. 이 방법은 다섯 개의 태스크가 동작하는 방법이며, 각 태스크마다 하나의 문제에 대해 채점하는 방식이다. 각 채점자마다 채점하는 문제에 대한 정보는 서로 다르다. 예를 들어, 문제 1은 셰익스피어에 대한 문제일 수 있고 문제 2는 밀턴에 대한 문제일 수 있다. 따라서 교수와 네 명의 조교는 서로 다른 채점 기준으로 채점을 하게 된다.

반대로, 두 번째 방법은 데이터 병렬화를 고려한 방법이다. "데이터"는 학생들이 제출한 답안지이며, 각 답안지를 코어의 수에 맞게 배분하여 각 코어들은 같은 채점 기준으로 채점하게 된다.

1.3절의 전역 합(global sum) 예제에서 첫 번째 부분은 데이터 병렬화의 예를 고려한 경우다. 데이터는 Compute_next_value로 계산되며 각 코어에서는 할당된 엘리먼트(element)만큼 같은 동작을 수행한다. Compute_next_value를 호출해서 주어진 값을 계산하고, 계산된 값을 합산한다. 첫 번째 전역 합의 두 번째 파트는 태스크 병렬화를 고려한 것이다. 여기서는 중간 합계를 받아서 각 코어의 중간 합계를 더하는 마스터 코어의 태스크와, 중간 합계를 계산하고 마스터 코어에 전송하고 마스터 코어 이외의 다른 코어에서 실행되는 태스크로 구성되어 있다.

코어들이 서로 독립적으로 실행할 때 병렬 프로그램은 시리얼 프로그램을 작성하는 방법과 거의 같다. 또한 코어들이 서로 협력할 필요가 있을 때 프로그램 작성 방법이 점점 더 복잡해지게 된다. 두 번째 전역 합의 예제에서 다이어그램의 트리 구조(tree-structure)는 이

해하기는 매우 쉽지만, 실제 코드를 작성하면 꽤 어렵다. 연습문제 1.3과 1.4를 참고하자. 불행하게도 각 코어들 간에 협력해야 할 필요가 훨씬 많다.

이 두 개의 전역 합 예제에서 코어들 간의 협력은 서로 간의 **통신**(communication)을 포함하고 있다. 하나 이상의 코어들이 현재의 부분 합을 다른 코어에게 전송한다. 전역 합의 예는 **로드 밸런싱**(load balancing)을 통한 협력도 포함하고 있다. 명시적인 수식으로 살펴보진 않았지만, 각 코어들마다 거의 비슷한 양의 계산을 하는 것이 바람직하다. 예를 들어, 하나의 코어가 대부분의 연산을 하고 나머지 코어들은 상대적으로 연산이 적어 빨리 끝난다면, 전체적인 연산 능력을 낭비하게 된다.

협력의 세 번째는 **동기화**(synchronization)다. 예를 들어, 연산할 값을 추가하는 대신에 stdin으로부터 그 값을 읽어 들인다고 가정해 보자. X는 마스터 코어에서 읽어 온 데이터의 배열이다.

```
if (마스터 코어라면)
   for (my_i = 0; my_i < n; my_i++)
      scanf("%lf", &x[my_i]);
```

대부분의 시스템에서 코어는 자동적으로 동기화하지 못한다. 오히려 각 코어는 다른 코어의 존재에 대한 고려 없이 동작한다. 이 경우에 문제는 다른 코어들이 아직 계산되지도 않은 중간 합계를 사용하려고 한다거나, 마스터 코어가 x를 초기화하고 다른 코어들에게 x를 사용해도 된다고 알리기도 전에 중간 합계를 계산하는 경우다. 이 말은 각 코어는 코드 실행 전에 기다려야 한다는 의미이다.

```
for (my_i = my_first_i; my_i < my_last_i; my_i++)
   my_sum += x[my_i];
```

x의 초기화와 중간 합계를 계산하는 코드 사이에 동기화를 추가해야 한다.

```
Synchrnonize_cores();
```

아이디어는 모든 코어가 연산과 관련된 함수를 실행하기 전에 Synchrnonize_cores 함수를 기다리도록 하는 것이며, 이 말의 의미는 마스터 코어가 이 함수를 실행할 때까지 기다린다는 의미이다.

현재 대부분의 병렬 프로그램은 **명시적(explicit)** 병렬 생성을 사용하며, C와 C++와 같은 프로그래밍 언어의 확장을 사용하여 작성한다. 이 프로그램은 병렬화에 대한 명시적 명령어를 포함하고 있다. 코어 0은 태스크 0을 실행하라, 코어 1은 태스크 1을 실행하라, . . . , 모든 코어는 동기화하라와 같은 명령어다. 이러한 프로그램은 상당히 복잡하다. 더욱이 현재의 다중 코어들로 인한 복잡도는 하나의 코어에서만 실행될 수 있는 프로그램을 작성하기 위해서 상당히 많은 것들을 고려해야 한다.

병렬 프로그램을 작성하기 위한 다른 방법이 있다. 예를 들어, 약간의 성능 향상은 포기하더라도 다소 쉽게 프로그램을 개발하는 상위 레벨 언어를 사용하는 것이다.

1.5 배울 내용

이제 명시적 병렬화 프로그램을 작성하는 방법에 대해 **구체적으로** 알아보기로 하자. 우리의 목적은 C 언어와 세 가지의 서로 다른 C 확장 기술인 **메시지 패싱 인터페이스(MPI, Message-Passing Interface)**, **POSIX threads** 혹은 **Pthreads**, 그리고 **OpenMP**를 사용하여 병렬 컴퓨터에서 프로그래밍하는 기본 개념에 대해 배워 보는 것이다. MPI와 Pthreads는 C 프로그램에서 사용할 수 있도록 타입 정의, 함수, 매크로 등으로 구성된 라이브러리이다. OpenMP는 라이브러리와 함께 약간의 C 컴파일러에 대한 수정도 필요하다.

위에서 언급한 기술 중에서 하나만 사용해도 될 것 같은데, 굳이 세 가지나 되는 C 확장 기술을 사용해야 하는 이유에 대해서 궁금해 하는 사람도 있을 것 같다. 이에 대한 해답은 각각의 확장 기술과 병렬 시스템에 있다. 우리가 눈여겨봐야 하는 병렬 시스템의 두 가지 중요한 타입은 **공유 메모리(shared-memory)** 시스템과 **분산 메모리(distributed-memory)** 시스템이다. 공유 메모리 시스템에서 코어는 컴퓨터의 메모리를 공유하며, 특히 각 코어는 각 메모리의 주소에 읽고 쓸 수 있다. 공유 메모리 시스템에서는 공유 메모리 주소를 액세스하거나 업데이트하면서 각 코어들 간의 협업을 할 수 있도록 한다. 분산 메모리 시스템에서는 반대로 각 코어는 자신만의 메모리, 즉 개인(private) 메모리를 갖고 있으며, 코어들 간에는 네트워크를 통해 메시지를 전송하는 등의 방법을 통해 명시적으로 통신해야 한다. 그림 1.2는 이 두 가지 시스템에 대한 구조를 보여 주고 있다. Pthreads와 OpenMP는 공

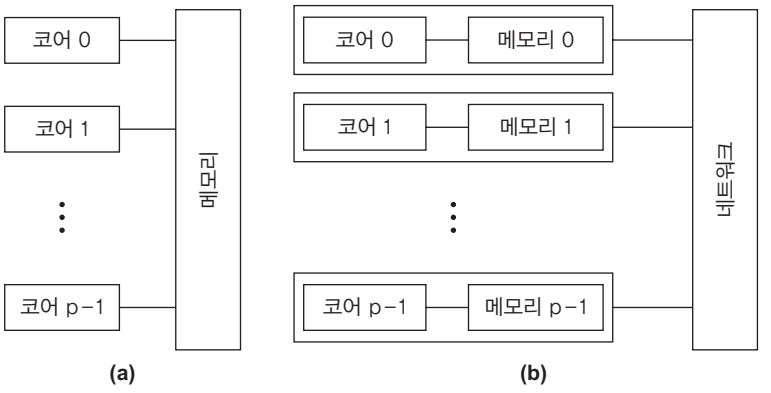

그림 1.2

(a) 공유 메모리 시스템 (b) 분산 메모리 시스템

유 메모리 시스템에서 사용되도록 설계됐다. 이 두 가지 기술은 공유 메모리 주소를 액세스할 수 있는 메커니즘을 제공하며, MPI는 분산 메모리 시스템에서 프로그래밍할 수 있도록 설계됐다. 따라서 메시지 전송과 관련된 메커니즘을 제공한다.

그런데 공유 메모리 시스템에서는 왜 두 가지 기술이 필요할까? OpenMP는 상대적으로 C 언어의 상위 레벨 확장이다. 예를 들면, 다음과 같은 루프(loop)를 하나의 디렉티브(directive)를 사용하여 "병렬화"할 수 있다.

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

반면에 Pthread를 사용하면 앞의 소스 코드와 유사한 형태의 구조를 갖게 된다. Pthread는 OpenMP에서 사용할 수 없는 몇 가지 더 로 레벨(low level)의 병렬화 기법을 제공한다. OpenMP를 사용하면 상대적으로 병렬화가 쉽지만, Pthreads는 다른 프로그램을 병렬화하기 쉽도록 해주는 기법을 제공한다.

1.6 병행, 병렬, 분산

병렬 컴퓨팅에 관한 여러 책을 보거나 웹에서 병렬 컴퓨팅에 대해 검색해 보면 **병행 컴퓨팅**(concurrent computing)과 **분산 컴퓨팅**(distributed computing)에 대한 용어도 만나게 된다. 병렬(parallel), 분산(distributed), 그리고 병행(concurrent)이라는 용어가 전혀 다른 용어는 아니지만, 많은 저자들은 다음과 같이 그 차이를 정의한다.

- 병행 컴퓨팅(concurrent computing)에서 하나의 프로그램은 일정한 짧은 기간에 여러 개의 태스크가 **동작**하는 것을 의미한다[4].
- 병렬 컴퓨팅(parallel computing)에서 하나의 프로그램은 어떤 문제를 해결하기 위해 여러 개의 태스크가 **밀접하게 연합하는** 것을 의미한다.
- 분산 컴퓨팅(distributed computing)에서 하나의 프로그램은 어떤 문제를 해결하기 위해 다른 프로그램과의 연합이 필요할 수 있다.

따라서 병렬 프로그램과 분산 프로그램은 병행(concurrent)한다고 말할 수 있으며, 멀티태스킹(multitasking) 운영체제와 같은 프로그램도 심지어 하나의 코어에서 동작하는 경우라 할지라도 병행한다고 말할 수 있다. 그 이유는 여러 태스크가 주어진 짧은 기간에서 **동작**하기 때문이다. 병렬과 분산은 확실한 차이로 나누기는 어렵지만, 병렬 프로그램은 보통 물리적으로 밀접한 다른 코어에서 여러 개의 태스크가 동시에 실행되는 것을 의미하며, 공유 메모리를 서로 공유하거나 고속 네트워크로 연결되어 있다. 한편, 분산 프로그램은 좀 더 "덜 밀접하게 연결되어" 있는 경향이 있다. 태스크는 물리적으로 많이 떨어진 여러 컴퓨터에서 실행되는 경우도 있으며, 태스크 자체도 서로 독립적으로 생성된 프로그램에 의해서 실행된다. 예를 들어, 두 개의 병행 덧셈 프로그램은 대부분의 개발자라면 병렬을 고려하겠지만, 웹 검색 프로그램의 경우라면 분산을 고려할 것이다.

이러한 용어가 일반적으로 통용되는 것은 아니다. 예를 들어, 공유 메모리 프로그램을 "병렬(parallel)"로 생각하고 분산 메모리 프로그램을 "분산(distributed)"이라고 생각하는 사람들도 많다. 이 책의 제목처럼 우리는 병렬 프로그램에 대해서 학습할 계획이고, 병렬 프로그램은 하나의 문제를 해결하기 위해서 여러 개의 밀접한 태스크들이 협력하는 프로그램을 의미한다.

1.7 책의 구성

이 책을 통해 병렬 프로그래밍을 할 때 어떤 도움을 받을 수 있는가?

첫 번째로, 고성능(high-performance)에 관심이 있고 시리얼 프로그램이냐 병렬 프로그램이냐 고민한다면 하드웨어와 소프트웨어가 동작하는 시스템 전반에 대해 알아야 한다. 2장에서 간략한 병렬 하드웨어와 병렬 소프트웨어에 관해 소개하고 있다. 이 내용을 이해하기 위해서는 시리얼 하드웨어와 소프트웨어의 몇 가지 중요한 내용도 살펴볼 필요가 있다. 하지만 초보자라면 2장의 대부분은 반드시 보지 않아도 되고, 이후의 내용을 읽으면서 필요할 때 참고하면 된다.

이 책의 핵심은 3장부터 6장까지이다. 3, 4, 5장은 C 언어를 사용한 병렬 프로그래밍과 MPI, Pthread, 그리고 OpenMP에 대해 가장 필수적인 내용을 소개하고 있다. 이 장을 읽기 전에 필요한 지식은 C 언어를 할 수 있느냐 정도다. 각각의 장은 서로 독립적이며 어떤 순서로 읽어도 상관없다. 그러나 각 장들을 독립적으로 구성하다 보니 몇 가지 개념은 중복되기도 한다. 세 장 중에서 하나의 장만 읽고 다른 장에서 중복된 내용은 넘어가고 새로운 내용만 읽어도 된다.

6장은 그 이후에 나올 장들에서 배울 내용을 합쳐 놓은 장이며, 공유 메모리와 분산 메모리 설정을 통해 두 개의 대규모 프로그램을 개발하는 내용에 대해 설명하고 있다. 그러나 3, 4, 5장 중에서 하나의 장만 선택해서 읽었다면 6장은 공부해야 할 내용이 좀 많을 수도 있다. 마지막으로 7장은 앞으로 병렬 프로그래밍에 관해 더 나은 학습을 하기 위한 몇 가지 제안을 하고 있다.

1.8 당부의 말

본격적으로 시작하기 전에 몇 가지 하고 싶은 이야기가 있다. 병렬 프로그램을 개발하는 일은 상당히 세심한 작업이며, 주의해서 설계하지 않으면 개발을 완료하기가 힘든 작업이다. 모든 병렬 프로그램은 적어도 하나의 시리얼 프로그램을 포함한다. 병렬 프로그램은

다중 코어의 동작을 주의 깊게 관리해야 해서 시리얼 프로그램보다 개발하기가 복잡하다. 사실 상당히 복잡하다. 주의를 기울여서 설계하고 개발해야 한다는 이러한 규칙은 시리얼 프로그램에서보다 병렬 프로그램에서 훨씬 중요하다.

1.9 이 책의 표기 방법

이 책에서 사용하는 몇 가지 표기 방법에 대해 알아보자.

- 프로그램 코드와 같은 경우는 다음과 같은 형식을 사용한다.

```
/* 간단한 프로그램 예제 */
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("hello, world\n");

   return 0;
}
```

- 용어에 대한 설명은 본문 안에 포함되어 있으며 그 본문 안에서 정의(definition)하는 용어는 굵은 서체로 나타냈다. 예를 들면, **병렬 프로그램**(parallel program)은 다중 코어를 사용한다.
- 프로그램이 개발되는 환경에 관해 언급할 필요가 있을 때는 유닉스 셸(UNIX shell)을 사용한다고 가정한다. 셸 프롬프트는 $ 문자로 표현한다.

```
$ gcc -g -Wall -o hello hello.c
```

- 고정된 매개변수 리스트를 사용하여 함수 호출에 대한 사용 방법을 정의할 수 있다. 예를 들어, 정수형 절댓값 함수인 abs는 stdlib 라이브러리에 다음과 같은 사용 방법으로 정의되어 있다.

```
int abs(int x); /* int x의 절댓값을 리턴한다 */
```

좀 더 복잡한 사용 방법의 경우는 꺾쇠 괄호(< >)를 사용하며, 옵션의 경우는 대괄호([ ])

를 사용한다. 예를 들어, C 언어에서 if 문장의 사용 방법은 다음과 같이 표현한다.

```
if ( <expression> )
   <statement1>
[else
   <statement2>]
```

위 문장은 if문은 반드시 괄호 안에 조건식 혹은 표현식이 있어야 하며, 조건식 혹은 표현식 다음의 닫는 괄호 다음에 statement1을 작성해야 한다. 또한, else 문장이 올 수도 있다. else 문장이 필요한 경우에는 두 번째 문장인 statement2가 있어야 한다.

1.10 요약

오랫동안 우리는 좀 더 빠른 프로세서를 사용해 왔다. 그러나 물리적인 한계로 인해 일반적인 프로세서를 사용한 성능 증가 비율이 점차 줄고 있다. 프로세서의 성능을 증가시키기 위해서 칩 제조업체들은 **멀티코어**(multicore) 통합 기판의 형태로 변경했으며, 이 통합 기판의 의미는 하나의 칩에 여러 개의 일반적인 프로세서를 집적했다는 의미다.

일반적인 **시리얼**(serial) 프로그램은 싱글 프로세서를 사용하도록 작성되었고, 따라서 다중 코어의 존재를 알지 못하기 때문에 프로그램을 변환해서 시리얼 코드를 **병렬화**(parallelizing)하는 것은 이점이 그다지 없다. 이 말은 기존의 시리얼 프로그램을 멀티코어가 사용 가능한 **병렬 프로그램**(parallel program)으로 변환해야 한다는 것을 의미한다.

병렬 프로그램을 개발할 때는 일반적으로 코어들 간의 작업을 분배하고 **관리**해야 한다. 코어들 간의 **통신**(communication), **로드 밸런싱**(load balancing), 그리고 코어들 간의 **동기화**(synchronization)를 포함하게 된다.

이 책을 통해 시스템의 성능을 극대화하기 위해 병렬 시스템을 프로그래밍하는 방법에 대해 배우게 된다. 앞으로 MPI, Pthreads 혹은 OpenMP를 C 언어와 함께 사용하는 방법을 배우게 된다. MPI는 **분산 메모리**(distributed-memory) 시스템에서 프로그래밍하는 데 사용되며, Pthreads와 OpenMP는 **공유 메모리**(shared-memory) 시스템에서 프로그래밍하는 데

사용된다. 분산 메모리 시스템에서의 코어들은 자신의 메모리를 갖고 있으며, 반면에 공유 메모리 시스템에서 코어들은 각 코어의 메모리를 액세스할 수 있다.

병행(concurrent) 프로그램은 어떤 짧은 기간에 여러 개의 태스크가 동작하는 것을 의미하며, **병렬(parallel)** 프로그램과 **분산(distributed)** 프로그램은 여러 개의 태스크가 동시에 동작하는 것을 의미한다. 병렬 프로그램에서 태스크들이 좀 더 밀접하게 동작하지만, 사실 병렬과 분산의 개념에는 큰 차이가 없다고 봐도 된다.

병렬 프로그램은 대개 상당히 복잡하다. 따라서 병렬 프로그램을 위해 적합한 프로그램 개발 기술을 사용하는 것이 중요하다.

1.11 연습문제

1.1 글로벌 합계 예제에서 my_first_i와 my_last_i를 계산하는 함수에 대한 공식을 만들어 보자. 각 코어는 루프에서 같은 양의 항목에 대한 계산을 할당받아야 한다는 것을 기억하자. (힌트) n이 p로 나뉠 수 있는 경우를 고려하자.

1.2 Compute_next에 대한 각각의 호출이 다른 호출과 마찬가지로 같은 양의 일이 필요하다는 것을 묵시적으로 가정한다. $i = 0$인 경우의 호출만큼이나 $i = k$에 대한 호출이 $k + 1$번 필요하다면 앞의 질문에 대한 답변은 어떻게 변할 수 있는가? 첫 번째 호출($i = 0$)이 2밀리세컨드가 걸리고, 두 번째 호출($i = 1$)이 4밀리세컨드가 걸리고, 세 번째 호출($i = 2$)은 6밀리세컨드가 걸린다고 가정한다.

1.3 그림 1.1에 있는 것처럼 트리 구조 글로벌 합계를 위한 의사 코드를 작성해 보자. 코어의 수는 2의 제곱(1, 2, 4, 8,...)이 된다. (힌트) 코어가 자신의 합계를 전송하는지 아니면 합계를 전송받아 덧셈하는지를 결정하기 위해 변수 divisor를 사용하자. divisor는 2로 시작하며 반복될 때마다 두 배씩 증가한다. 또한 core_difference 변수를 사용하여 어떤 코어가 현재 코어와 파트너로 작업하는지를 결정한다. 이 변수는 1을 갖고 시작하며 마찬가지로 반복될 때마다 두 배로 증가한다. 예를 들어, 첫 번

째 반복 0 % divisor = 0이고 1 % divisor = 1인 경우에, 1이 전송할 때 0은 수신받아서 덧셈을 하게 된다. 또한 첫 번째 반복은 0 + core_difference = 1이고 1 − core_difference = 0인 경우에 0과 1은 첫 번째 반복에서 한 쌍으로 간주된다.

1.4 앞의 문제에서 설명한 방법과 다른 방법으로 C의 비트와이즈(bitwise) 연산자를 사용하여 트리 구조 글로벌 합계를 구현할 수 있다. 이 작업이 동작하는 방법을 알기 위해서 각 코어의 랭크에 대한 표현으로 바이너리(base는 2다)를 작성해 보고 각 단계마다 어떻게 한 쌍이 되는지 적어 보자.

코어	단계		
	1	2	3
$0_{10} = 000_2$	$1_{10} = 001_2$	$2_{10} = 010_2$	$4_{10} = 100_2$
$1_{10} = 001_2$	$0_{10} = 000_2$	×	×
$2_{10} = 010_2$	$3_{10} = 011_2$	$0_{10} = 000_2$	×
$3_{10} = 011_2$	$2_{10} = 010_2$	×	×
$4_{10} = 100_2$	$5_{10} = 101_2$	$6_{10} = 110_2$	$0_{10} = 000_2$
$5_{10} = 101_2$	$4_{10} = 100_2$	×	×
$6_{10} = 110_2$	$7_{10} = 111_2$	$4_{10} = 100_2$	×
$7_{10} = 111_2$	$6_{10} = 110_2$	×	×

표를 보면 첫 번째 단계에서는 가장 오른쪽 혹은 첫 번째 비트가 다른 랭크를 가진 코어끼리 한 쌍으로 묶였다는 것을 알 수 있다. 두 번째 단계에서는 두 번째 비트가 다른 랭크의 코어들끼리 한 쌍으로 묶였다는 것을 알 수 있다. 세 번째는 세 번째 비트가 다른 랭크의 코어들끼리 묶였다. 따라서 첫 번째 단계에서 bitmask의 바이너리 값이 001_2를 갖고 있다면 두 번째에는 010_2 그리고 세 번째는 100_2가 된다. bitmask에서 0이 아닌 값을 갖는 랭크의 비트를 "인버트(inverting)"하여 쌍으로 묶을 수 있는 코어들의 랭크를 구할 수 있다. 이 작업은 비트와 이즈의 exclusive or 혹은 ^ 연산자를 사용하면 된다. 비트 exclusive or와 왼쪽 시프트(left-shitf) 연산자를 사용하여 의사 코드로 이 알고리즘을 구현해 보자.

1.5 연습문제 1.3 혹은 연습문제 1.4의 의사 코드(pseudo-code)가 코어의 수가 2의 제곱이 아닌 값(예를 들어, 3, 5, 6, 7)을 갖는다면 어떤 일이 발생하는가? 의사코드를 수정해서 코어의 수와 상관없이 올바르게 동작하도록 할 수 있는가?

1.6 수신받는 수와 코어 0이 다음을 사용할 때의 수식을 구해 보자.

a. 글로벌 합계를 위한 원래의 의사코드

b. 트리 구조 글로벌 합계

두 합계가 2, 4, 8, ..., 1024 코어에서 사용될 때 코어 0에 의해 실행되는 덧셈과 수신하는 수를 보이는 테이블을 작성해 보자.

1.7 글로벌 합계 예제의 첫 번째 파트는 각 코어가 할당된 계산된 값을 덧셈할 때 데이터 병렬화의 예제로 보통 간주된다. 첫 번째 합계의 두 번째 파트는 코어가 자신의 부분 합계를 마스터 코어에 전송할 때 그 값을 덧셈한다. 이것은 태스크 병렬화의 예제로 간주된다. 코어가 자신의 부분 합계를 덧셈하기 위해 트리 구조를 사용할 때 두 번째 글로벌 합계의 두 번째 부분은 어떻게 될까? 이것이 데이터 병렬화일까? 아니면 태스크 병렬화일까? 그 이유는 무엇인가?

1.8 교수가 학부의 학생들을 위해 파티를 연다고 가정해 보자.

a. 교수가 파티를 준비할 때 태스크 병렬화를 사용하기 위해 허용된 교수의 수를 할당받은 태스크를 작성하자. 다양한 태스크가 실행될 때의 스케줄을 작성해 보자.

b. 앞 부분에서 태스크 중 하나는 파티가 개최될 장소를 청소해야 한다고 가정하자. 데이터 병렬화를 사용하여 교수들 중에 파티 장소를 청소하도록 작업을 파티셔닝하려면 어떻게 해야 할까?

c. 파티를 준비하기 위해 태스크 병렬화와 데이터 병렬화를 조합하여 사용하자(교수들에게 일이 너무 많으면 조교를 뽑아서 사용할 수 있다).

1.9 여러분 전공의 여러 문제들에서 병렬 컴퓨팅을 사용할 때 얻을 수 있는 장점에 대해 에세이를 작성해 보자. 어떻게 병렬화를 사용할 수 있는지 간략하게 작성해 보자. 태스크 병렬화를 사용할 것인가? 아니면 데이터 병렬화를 사용할 것인가?