(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2018년09월20일
(11) 등록번호 10-1900796
(24) 등록일자 2018년09월14일

(51) 국제특허분류(Int. Cl.)
G06F 9/38 (2006.01) *G06F 9/44* (2018.01)
G06F 9/46 (2006.01)
(21) 출원번호 10-2013-7016249
(22) 출원일자(국제) 2011년12월20일
심사청구일자 2016년11월18일
(85) 번역문제출일자 2013년06월21일
(65) 공개번호 10-2014-0014090
(43) 공개일자 2014년02월05일
(86) 국제출원번호 PCT/US2011/066285
(87) 국제공개번호 WO 2012/088174
국제공개일자 2012년06월28일
(30) 우선권주장
12/975,796 2010년12월22일 미국(US)
(56) 선행기술조사문헌
US20100199257 A1
US20050071828 A1

(73) 특허권자
마이크로소프트 테크놀로지 라이센싱, 엘엘씨
미국 워싱턴주 (우편번호 : 98052) 레드몬드 원 마이크로소프트 웨이
(72) 발명자
링세스 폴 에프
미국 워싱턴주 98052-6399 레드몬드 원 마이크로소프트 웨이 엘씨에이 - 인터내셔널 페이턴츠 마이크로소프트 코포레이션
(74) 대리인
제일특허법인(유)

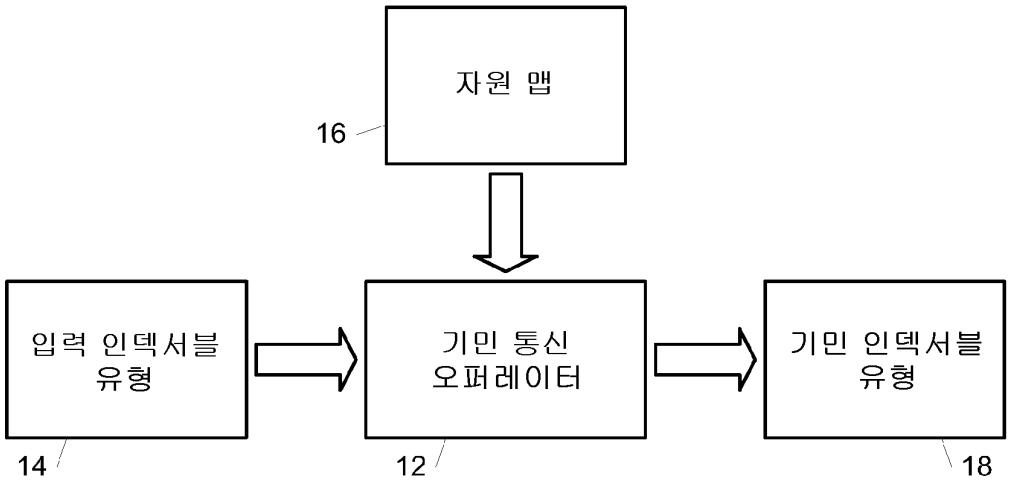
전체 청구항 수 : 총 20 항 심사관 : 정성훈

(54) 발명의 명칭 **기민 통신 오퍼레이터**

(57) 요 약

고레벨 프로그래밍 언어는 컴퓨트 노드들에 걸쳐 계산 공간을 분배하기 위한 자원 맵에 기초하여 세그먼트화된 계산 공간을 생성하는 기민 통신 오퍼레이터를 제공한다. 기민 통신 오퍼레이터는 통신 공간을 세그먼트들로 분해하고, 세그먼트들이 컴퓨트 노드들에 할당되도록 하며, 이용자가 컴퓨트 노드들 사이에서의 세그먼트들의 이동을 중앙 집중식으로 관리 및 자동화하는 것을 가능하게 한다. 세그먼트 이동은 세그먼트들의 전체 전역-뷰 표현 또는 국지적-전역 뷰 표현을 이용하여 관리될 수 있다.

대 표 도



명 세 서

청구범위

청구항 1

컴퓨터 시스템에서 컴파일러에 의해 수행되는 방법으로서,

특수 목적 아키텍처를 가진 하나 이상의 데이터 병렬 최적 컴퓨트 노드 상에서의 실행을 위해 구성되는 데이터 병렬 소스 코드 내에서 기민 통신 오퍼레이터(agile communication operator)를 식별하는 단계 - 상기 데이터 병렬 소스 코드는, 범용 프로세서를 이용하는 것보다 더 빠르게 혹은 더 효율적으로 데이터 병렬 동작이 실행되도록, 상기 데이터 병렬 최적 컴퓨트 노드의 상기 특수 목적 아키텍처의 장점을 취하는 데이터 병렬 피처(feature)를 포함함 - 와,

데이터 병렬 실행 가능 코드가, 입력 인덱서블 유형 및 자원 맵으로부터 기민 인덱서블 유형(agile indexable type)을 생성함으로써, 상기 기민 통신 오퍼레이터를 구현하도록, 상기 데이터 병렬 소스 코드로부터 상기 데이터 병렬 실행 가능 코드를 생성하는 단계 - 상기 기민 통신 오퍼레이터는 상기 데이터 병렬 최적 컴퓨트 노드에 걸쳐서 세그먼트화된 계산 공간을 분산시키도록 상기 자원 맵에 기초해서 상기 세그먼트화된 계산 공간을 생성함 -

을 포함하는 컴파일러에 의해 수행되는 방법.

청구항 2

제 1 항에 있어서,

상기 기민 인덱서블 유형은 상기 자원 맵에 의해 규정되는 복수의 세그먼트를 가지는

컴파일러에 의해 수행되는 방법.

청구항 3

제 1 항에 있어서,

상기 데이터 병렬 실행 가능 코드가, 상기 하나 이상의 데이터 병렬 최적 컴퓨트 노드 중 제 1 노드 상에서 상기 기민 인덱서블 유형의 제 1 세그먼트를 이용하고, 상기 하나 이상의 데이터 병렬 최적 컴퓨트 노드 중 제 2 노드 상에서 상기 기민 인덱서블 유형의 제 2 세그먼트를 이용함으로써, 상기 기민 통신 오퍼레이터를 구현하도록, 상기 데이터 병렬 소스 코드로부터 상기 데이터 병렬 실행 가능 코드를 생성하는 단계를 더 포함하는

컴파일러에 의해 수행되는 방법.

청구항 4

제 1 항에 있어서,

상기 데이터 병렬 소스 코드는 상기 기민 인덱서블 유형의 전체 전역-뷰 표현(a full global-view representation) 으로 코딩되는

컴파일러에 의해 수행되는 방법.

청구항 5

제 1 항에 있어서,

상기 데이터 병렬 소스 코드는 상기 기민 인덱서블 유형의 국지적-전역-뷰 표현(a local-global-view representation) 으로 코딩되는

컴파일러에 의해 수행되는 방법.

청구항 6

제 1 항에 있어서,

상기 데이터 병렬 소스 코드는 데이터 병렬 확장을 가지는 고레벨 범용 프로그래밍 언어로 기록되는

컴파일러에 의해 수행되는 방법.

청구항 7

제 1 항에 있어서,

상기 데이터 병렬 소스 코드는 고레벨 데이터 병렬 프로그래밍 언어로 기록되는

컴파일러에 의해 수행되는 방법.

청구항 8

제 1 항에 있어서,

상기 하나 이상의 데이터 병렬 컴퓨트 노드는 적어도 하나의 그래픽 처리 장치를 포함하는

컴파일러에 의해 수행되는 방법.

청구항 9

제 1 항에 있어서,

상기 하나 이상의 데이터 병렬 컴퓨트 노드는 적어도 하나의 범용 프로세서를 포함하는

컴파일러에 의해 수행되는 방법.

청구항 10

컴퓨터 실행 가능 명령이 저장된 컴퓨터 판독 가능 저장 메모리로서,

상기 컴퓨터 실행 가능 명령은, 컴퓨터 시스템에 의해 실행될 때, 상기 컴퓨터 시스템으로 하여금,

특수 목적 아키텍처를 가진 하나 이상의 데이터 병렬 최적 컴퓨트 노드 상에서의 실행을 위해 구성되는 데이터 병렬 소스 코드 내의 기민 통신 오퍼레이터에 응답하여 입력 인덱서블 유형 및 자원 맵으로부터 기민 인덱서블 유형을 생성하는 단계 - 상기 데이터 병렬 소스 코드는, 범용 프로세서를 이용하는 것보다 더 빠르게 혹은 더 효율적으로 데이터 병렬 동작이 실행되도록, 상기 데이터 병렬 최적 컴퓨트 노드의 상기 특수 목적 아키텍처의 장점을 취하는 데이터 병렬 피처를 포함하고, 상기 기민 통신 오퍼레이터는 상기 데이터 병렬 최적 컴퓨트 노드에 걸쳐서 세그먼트화된 계산 공간을 분산시키도록 상기 자원 맵에 기초해서 상기 세그먼트화된 계산 공간을 생성함 - 와,

상기 기민 인덱서블 유형을 사용해서 데이터 병렬 알고리즘을 수행하는 단계

를 포함하는 방법을 수행하게 하는

컴퓨터 판독 가능 저장 메모리.

청구항 11

제 10 항에 있어서,

상기 기민 인덱서블 유형은 상기 자원 맵에 의해 규정되는 복수의 세그먼트를 가지는

컴퓨터 판독 가능 저장 메모리.

청구항 12

제 10 항에 있어서,

상기 방법은,

하나 이상의 데이터 병렬 최적 컴퓨트 노드 중 제 1 노드 상에서 상기 기민 인덱서블 유형의 제 1 세그먼트를 이용하고, 상기 하나 이상의 데이터 병렬 최적 컴퓨트 노드 중 제 2 노드 상에서 상기 기민 인덱서블 유형의 제 2 세그먼트를 이용하여, 상기 데이터 병렬 알고리즘을 수행하는 단계

를 더 포함하는

컴퓨터 판독 가능 저장 메모리.

청구항 13

제 10 항에 있어서,

상기 데이터 병렬 알고리즘은 상기 기민 인덱서블 유형의 전체 전역-뷰 표현으로 코딩되는

컴퓨터 판독 가능 저장 메모리.

청구항 14

제 10 항에 있어서,

상기 데이터 병렬 알고리즘은 상기 기민 인덱서블 유형의 국지적-전역-뷰 표현으로 코딩되는

컴퓨터 판독 가능 저장 메모리.

청구항 15

제 10 항에 있어서,

상기 데이터 병렬 소스 코드는 데이터 병렬 확장을 가지는 프로그래밍 언어로 기록되는

컴퓨터 판독 가능 저장 메모리.

청구항 16

제 10 항에 있어서,

상기 데이터 병렬 소스 코드는 고레벨 데이터 병렬 프로그래밍 언어로 기록되는

컴퓨터 판독 가능 저장 메모리.

청구항 17

제 10 항에 있어서,

상기 하나 이상의 데이터 병렬 최적 컴퓨트 노드는 적어도 하나의 그래픽 처리 장치를 포함하는

컴퓨터 판독 가능 저장 메모리.

청구항 18

제 10 항에 있어서,

상기 하나 이상의 데이터 병렬 최적 컴퓨트 노드는 적어도 하나의 범용 프로세서를 포함하는

컴퓨터 판독 가능 저장 메모리.

청구항 19

컴퓨터 실행 가능 명령이 저장된 컴퓨터 판독 가능 저장 메모리로서,

상기 컴퓨터 실행 가능 명령은, 컴퓨터 시스템에 의해 실행될 때, 상기 컴퓨터 시스템으로 하여금,

자원 맵에 따라서 제 1 랭크 및 제 1 요소 유형(element type)을 가진 입력 인덱서블 유형에 기민 통신 오퍼레이터를 적용해서, 복수의 세그먼트를 가진 기민 인덱서블 유형을 생성하는 단계와,

상기 기민 인덱서블 유형을 이용해서 데이터 병렬 알고리즘을 수행하는 단계

를 포함하는 방법을 수행하게 하고,

상기 기민 통신 오퍼레이터는 데이터 병렬 소스 코드에 포함되되, 상기 데이터 병렬 소스 코드는, 범용 프로세서를 이용하는 것보다 더 빠르게 혹은 더 효율적으로 데이터 병렬 동작이 실행되도록, 데이터 병렬 최적 컴퓨트 노드의 특수 목적 아키텍처의 장점을 취하는 데이터 병렬 확장(data parallel extensions)을 가지는 고레벨 범용 프로그래밍 언어로 기록되며,

상기 기민 통신 오퍼레이터는 상기 데이터 병렬 최적 컴퓨트 노드에 걸쳐서 세그먼트화된 계산 공간을 분산시키도록 상기 자원 맵에 기초해서 상기 세그먼트화된 계산 공간을 생성하는

컴퓨터 판독 가능 저장 메모리.

청구항 20

제 19 항에 있어서,

상기 데이터 병렬 소스 코드는 적어도 하나의 그래픽 처리 장치를 포함하는 상기 하나 이상의 데이터 병렬 최적 컴퓨트 노드에서 실행하기 위해서 구성되는

컴퓨터 판독 가능 저장 메모리.

발명의 설명

기 술 분 야

배 경 기 술

[0001] 컴퓨터 시스템들은 흔히 하나 이상의 범용 프로세서들(예를 들어, 중앙 처리 장치(CPU)들) 및 하나 이상의 특화 데이터 병렬 컴퓨트 노드(compute node)들(예를 들어, 그래픽 프로세싱 장치(GPU)들 또는 CPU들 내의 단일 명령, 다중 데이터(single instruction, multiple data; SIMD) 수행 장치들)을 포함한다. 범용 프로세서들은 일반적으로 컴퓨터 시스템들 상에서 범용 프로세싱을 수행하고, 데이터 병렬 컴퓨트 노드들은 일반적으로 컴퓨터 시스템들 상에서 데이터 병렬 프로세싱(예를 들어, 그래픽 프로세싱)을 수행한다. 범용 프로세서들은 흔히 데이터 병렬 알고리즘들을 구현하는 능력을 가지고 있지만, 데이터 병렬 컴퓨트 노드들에서 발견되는 최적화된 하드웨어 자원들 없이 이를 행한다. 결과적으로, 범용 프로세서들은 데이터 병렬 알고리즘을 수행하는 데 있어서 병렬 컴퓨트 노드들보다 효율이 훨씬 더 적을 수 있다.

[0002] 데이터 병렬 컴퓨트 노드들은 종래에는 컴퓨터 시스템들 상에서 프로그램들을 수행하는 데 있어서 범용 프로세서들을 지원하는 역할을 해왔다. 데이터 병렬 알고리즘들에 대해 최적화된 하드웨어의 역할이 데이터 병렬 컴퓨트 노드 프로세싱 케이퍼빌리티(capability)들의 강화로 인해 증가함에 따라, 프로그래머들이 데이터 병렬 컴퓨트 노드들을 프로그램하고 데이터 병렬 컴퓨트 노드들의 프로그래밍을 더 쉽게 하는 능력을 향상시키는 것이 바람직할 것이다.

[0003] 데이터 병렬 알고리즘들은 흔히 다수의 컴퓨팅 플랫폼들에 걸쳐 분포될 수 있는 거대한 데이터의 세트들 상에서 동작한다. 거대한 데이터의 세트들로 인해 데이터를 기술하는 데이터 구조들을 표현하고 추적하는 데 있어서뿐만 아니라 데이터를 다수의 플랫폼들에 걸쳐 이동시키는 데 있어서 어려움들이 있다. 결과적으로, 다수의 컴퓨팅 플랫폼들에 걸친 거대한 데이터의 세트들을 관리하는 프로세스는 흔히 복잡하고 구현하기 어렵다.

발명의 내용

해결하려는 과제

과제의 해결 수단

[0004] 이 요약은 상세한 설명에서 더 후술되어 있는 간소화된 형태의 개념들의 선택을 도입하도록 제공된다. 이 요약은 청구되는 특허대상의 핵심 특징들 또는 필수적인 특징들을 식별하도록 의도되거나, 청구되는 특허 대상의 범위를 제한하는 데 이용되도록 의도되지 않는다.

[0005] 고레벨 프로그래밍 언어는 컴퓨트 노드들에 걸쳐 계산 공간을 분배하기 위한 자원 맵에 기초하여 세그먼트화된 계산 공간을 생성하는 기민 통신 오퍼레이터를 제공한다. 기민 통신 오퍼레이터는 계산 공간을 세그먼트들로 분해하고, 세그먼트들이 컴퓨트 노드들에 할당되도록 하며, 이용자가 컴퓨트 노드들 사이에서의 세그먼트들의 이동을 중앙집중식으로 관리 및 자동화하는 것을 가능하게 한다. 세그먼트 이동은 세그먼트들의 전체 전역-뷰 표현 또는 국부적 전역 뷰 표현을 이용하여 관리될 수 있다.

도면의 간단한 설명

[0006] 첨부 도면들은 실시예들의 심도 있는 이해를 제공하기 위해 포함되고 본 명세서의 일부에 통합되고 일부를 구성한다. 도면들은 실시예들을 도시하고 설명과 함께 실시예들의 원리들을 설명하는 역할을 한다. 다른 실시예들 및 실시예들의 의도된 장점들의 대다수는 다음의 상세한 설명을 참조하여 더 양호하게 이해될 때 순조롭게 인식될 수 있다. 도면들의 요소들은 반드시 서로에 대해 비례하는 것은 아니다. 동일한 참조 번호들은 대응하는 유사한 파트들을 지정한다.

도 1은 기민 통신 오퍼레이터를 지니는 코드의 실시예를 도시하는 컴퓨터 코드도;

도 2는 기민 통신 오퍼레이터를 입력 인덱서블 유형(indexable type)에 적용하는 실시예를 도시하는 블록도;

도 3a 내지 도 3c는 기민 인덱서블 유형을 생성하고 이용하는 예들을 도시하는 블록도들;

도 4는 기민 통신 오퍼레이터를 포함하는 데이터 병렬 코드를 컴파일링하고 수행하도록 구성되는 컴퓨터 시스템의 실시예를 도시하는 블록도.

발명을 실시하기 위한 구체적인 내용

[0007] 다음의 상세한 설명에서, 본 발명의 일부를 형성하고 본 발명이 실행될 수 있는 설명 지정 실시예들에 의해 도시되는 첨부 도면들이 참조된다. 이 점에 있어서, "상부", "하부", "전방", "후방", "앞서는", "뒤처지는" 등과 같은 방향 용어는 기술되는 도면(들)의 방위에 관하여 이용된다. 실시예들의 구성요소들은 다양한 방위들로 포지셔닝할 수 있으므로, 방향 용어는 설명을 위해 이용되며 결코 제한하지 않는다. 본 발명의 범위를 벗어나지 않고 다른 실시예들이 이용되고 구조 또는 논리 변화들이 행해질 수 있음이 이해될 것이다. 그러므로, 다음의 상세한 설명은 제한하는 의미로 취해지지 않아야 하며, 본 발명의 범위는 첨부된 청구항들에 의해서만 규정된다. 본원에서 기술되는 다양한 예시적인 실시예들의 특징들은 달리 구체적으로 지적되지 않는 한, 서로 결합될 수 있음이 이해되어야 한다.

[0008] 도 1은 기민 통신 오퍼레이터(12)를 지니는 코드(10)의 실시예를 도시하는 컴퓨터 코드도이다. 컴파일링되고 수행될 때, 기민 통신 오퍼레이터(12)는 컴퓨트 노드들(예를 들어, 도 4에 도시되고 추가적으로 더 상세하게 후술되는 컴퓨트 노드들(121)에 걸쳐 계산 공간(computational space)을 분배하기 위해 자원 맵(resource map)에 기초하여 분절화된 계산 공간을 생성한다. 기민 통신 오퍼레이터는 계산 공간(도 1의 실시예에서 입력 인덱서블 유형(14)에 의해 표현된다)을 기민 인덱서블 유형(18)(또한 도 3b의 예에서 도시된다)의 세그먼트들(20)로 분해하고, 세그먼트들(20)이 컴퓨트 노드들로 할당되도록 하고 이용자가 중앙집중식으로 컴퓨트 노드들 사이의 세그먼트들(20)의 이동을 관리하고 자동화하도록 한다. 세그먼트 이동은 추가적으로 상세하게 후술되는 바와 같이 세그먼트들의 전체 전역-뷰(global view) 표현 또는 국부적 전역 뷰 표현을 이용하여 관리될 수 있다.

[0009] 코드(10)는 하나 이상의 DP 최적 컴퓨트 노드들(예를 들어, 도 4에 도시된 DP 최적 컴퓨트 노드들(121))에 의해 수행되는 하나 이상의 실행 파일(executable)들(예를 들어 도 4에 도시된 DP 실행파일(138))로 컴파일링될 수 있는 고레벨 범용 또는 데이터 병렬 프로그래밍 언어로부터의 명령들의 스퀀스를 포함한다.

[0010] 하나의 실시예에서, 코드(10)는 하나 이상의 모듈들의 세트에 저장되는 프로그램을 형성하는 데이터 병렬 확장들을 가지는 고레벨 범용 프로그래밍 언어(이후에는 GP 언어)로부터의 명령들의 시퀀스를 포함한다. GP 언어는 프로그램이 상이한 파트들(즉, 모듈들)에 기록되는 것이 가능하게 함으로써, 각각의 모듈은 컴퓨터 시스템에 의해 액세스될 수 있는 별개의 파일들 또는 위치들에 저장될 수 있게 된다. GP 언어는 하나 이상의 범용 프로세서들 및 하나 이상의 특수 목적, DP 최적 컴퓨트 노드들을 포함하는 컴퓨팅 환경을 프로그래밍하기 위하여 단일 언어를 제공한다. DP 최적 컴퓨트 노드들은 전형적으로 범용 프로세서들의 그래픽 처리 장치(GPU)들 또는 SIMD 장치들이지만 또한 범용 프로세서들, 필드 프로그래머블 게이트 어레이(FPGA)들 또는 어떤 컴퓨터 환경들에 있는 다른 적절한 디바이스들의 스칼라 또는 벡터 실행 장치들을 포함할 수 있다. GP 언어를 이용함으로써 프로그래머는 범용 프로세서들 및 DP 컴퓨트 노드들에 의한 실행을 위해 코드(10)에 범용 프로세서 및 DP 소스 코드 모두를 포함하고 범용 프로세서 및 DP 코드의 실행을 조정할 수 있다. 코드(10)는 애플리케이션, 라이브러리 함수 또는 운영 시스템 서비스와 같이, 본 실시예에서 임의의 적합한 코드 유형을 나타낼 수 있다.

[0011] GP 언어는 데이터 병렬 피처(feature)들을 포함하기 위해 폭넓게 적응되는 C 또는 C++와 같은 고레벨 및 범용 프로그래밍 언어를 확장함으로써 형성될 수 있다. DP 피처들이 나타날 수 있는 다른 범용 언어들의 예들은 Java™, PHP, Visual Basic, Perl, Python™, C#, Ruby, Delphi, Fortran, VB, F#, OCaml, Haskell, Erlang, NESL, Chapel, JavaScript™을 포함한다. GP 언어 구현은 프로그램의 상이한 부분들이 상이한 모듈들에 포함되도록 하는 풍부한 링크 케이퍼빌리티(capability)들을 포함할 수 있다. 데이터 병렬 피처들은 데이터 병렬 연산들이 범용 프로세서들(즉, 비-DP 최적 컴퓨트 노드들)에 있어서보다 더 빠르고 더 효율적으로 수행되도록 하는 DP 최적 컴퓨트 노드들의 특수 목적 아키텍처의 장점을 취하는 프로그래밍 툴들을 제공한다. GP 언어는 또한 프로그래머가 범용 프로세서들 및 DP 최적 컴퓨트 노드들 모두에 대해 프로그램하는 것이 가능한 다른 적절한 고레벨 범용 프로그래밍 언어일 수 있다.

[0012] 다른 실시예에서, 코드(10)는 프로그램을 형성하는 고레벨 데이터 병렬 프로그래밍 언어(이후에는 DP 언어)로부터의 명령들의 시퀀스를 포함한다. DP 언어는 하나 이상의 DP 최적 컴퓨트 노드들을 구비하는 컴퓨팅 환경에서 DP 최적 컴퓨트 노드를 프로그래밍하는 특수 언어를 제공한다. DP 언어를 이용하여, 프로그래머는 DP 최적 컴퓨트 노드들에 대해 실행하고자 하는 코드(10)에 DP 소스 코드를 생성한다. DP 언어는 데이터 병렬 연산들이 범용 프로세서들에 있어서보다 더 빠르고 더 효율적으로 수행되도록 하는 DP 최적 컴퓨트 노드들의 특수 목적 아키텍처의 장점을 취하는 프로그래밍 툴들을 제공한다. DP 언어는 HLSL, GLSL, Cg, C, C++, NESL, Chapel, CUDA, OpenCL, Accelerator, Ct, PGI GPGPU 가속기, CAPS GPGPU 가속기, Brook+, CAL, APL, Fortran 90 (및 상위), Data Parallel C, DAPPLE 또는 APL과 같은 기존 DP 프로그래밍 언어일 수 있다. 코드(10)는 애플리케이션, 라

이브러리 함수 또는 운영 시스템 서비스와 같이, 본 실시예에서 임의의 적합한 DP 소스 코드 유형을 나타낼 수 있다.

[0013] 코드(10)는 DP 최적 컴퓨트 노드 상에서의 실행을 위해 지정되는 코드 부분들을 포함한다. 코드(10)가 GP 언어로 기록되는 도 1의 실시예에서, GP 언어는 프로그래머가 벡터 함수를 규정할 때 주석(26)(예를 들어, declspec(vector)…)을 이용하여 DP 소스 코드를 지정하도록 한다. 주석(26)은 DP 최적 컴퓨트 노드 상에서의 실행을 위해 의도되는 벡터 함수의 함수 이름(27)(예를 들어, vector func)과 연관된다. 코드(10)는 또한 콜 사이트(call site)(예를 들어, forall, reduce, scan 또는 sort)에서의 벡터 함수(예를 들어, forall…, vector func,…)의 하나 이상의 호출(invocation)(28)을 포함할 수 있다. 콜 사이트에 대응하는 벡터 함수는 커널 함수로서 칭해진다. 커널 함수는 코드(10) 내의 다른 벡터 함수들(즉, 다른 DP 소스 코드)을 호출할 수 있고, 벡터 함수 콜 그래프(call graph)의 루트로서 간주될 수 있다. 커널 함수는 또한 코드(10)에 의해 규정되는 유형들(예를 들어, class들 또는 struct들)을 이용할 수 있다. 유형들은 DP 소스 코드로서 주석 첨가될 수 있거나 되지 않을 수 있다. 다른 실시예들에서, 코드(10)의 부분들을 DP 소스 코드 및/또는 범용 프로세서 코드로서 지정되도록 다른 적절한 프로그래밍 언어 구성들이 이용될 수 있다. 게다가, 주석들(26)은 코드(10)가 DP 언어로 기록되는 실시예들에서 생략될 수 있다.

[0014] 도 2는 기민 인덱서블 유형(18)을 제작하기 위해 입력 인덱서블 유형(14)에 기민 통신 오퍼레이터(12)를 적용하는 실시예를 도시하는 블록도이다. 본원에서 이용되는 바와 같이, 인덱서블 유형은 음이 아닌 정수인 랭크 및 element_type으로 표시되는 유형과 함께 하나 이상의 서브스크립 오퍼레이터들을 구현하는 임의의 데이터 유형이다. Index<N>이 정수들의 N-튜플(tuple)들을 나타내는 유형(즉, 정수 데이터 유형 중 임의의 유형)인 경우, index<N>의 인스턴스(instance)는 N 정수들의 세트이고{i0, i1,…,im}, 여기서 m은 N-1과 같다(즉, N-튜플). 랭크 N의 인덱스 오퍼레이터는 index<N>의 N-튜플 인스턴스를 취하고 이 인스턴스를 요소 유형이라 칭해지는 유형의 다른 인스턴스와 연관시키며 여기서 요소 유형은 인덱서블 유형 내의 각각의 요소를 규정한다. 하나의 실시예에서, 인덱서블 유형은 다음의 오퍼레이터들 중 하나 이상을 규정한다:

```
element_type operator[] (index_declarator);
 const element_type operator[] (index_declarator) const;
 element_type& operator[] (index_declarator);
 const element_type& operator[] (index_declarator) const;
  element_type&& operator[] (index_declarator); or
  const element_type&& operator[] (index_declarator)
const;
```

[0015]

[0016] 여기서 index_declarator는 다음 중 적어도 하나의 형태를 취한다:

```
const index<rank>& idx;
const index<rank> idx;
index<rank>& idx;
index<rank> idx.
```

[0017]

[0018] 다른 실시예들에서, 오퍼레이터들은 함수들, 기능어(functor)들 또는 더 일반적인 표현일 수 있다. 인덱서블 유형의 형상은 상기 서브스크립 오퍼레이터들 중 하나가 규정되는 index<rank>의 세트이다. 인덱서블 유형은 전형적으로 폴리토프(polytope)인 유형을 가진다. 즉, 인덱서블 유형은 좌표축들의 선형 함수들에 의해 형성되는 유한한 수효의 반-공간(half-space)들의 교차점으로서 대수적으로 표현될 수 있다.

[0019] 도 1 및 도 2를 참조하면, 코드(10)의 고레벨 언어는 하나의 실시예에서 데이터 병렬 컴퓨팅 환경에서 입력 인덱서블 유형(14)에 대해 이용하기 위한 기민 통신 오퍼레이터(12)를 제공한다. 입력 인덱서블 유형(14)은 랭크(예를 들어, 도 1의 실시예에서 랭크(N)) 및 요소 유형(예를 들어, 도 1의 실시예에서 요소 유형(T))을 가지며 기민 통신 오퍼레이터(12)에 의해 현재 동작 중인 계산 공간을 규정한다. 기민 통신 오퍼레이터(12)는 입력 인덱서블 유형(14) 및 자원 맵(16)(예를 들어, 도 1의 예에서의 자원 맵)을 수신한다. 입력 인덱서블 유형(14) 및 자원 맵(16)으로부터, 기민 통신 오퍼레이터(12)는 자원 맵(16)(또한 도 3b의 예에서 도시됨)에 의해 명시되고, 또한 서브-그리드들로 칭해지는 세그먼트들(20)을 가지는 기민 인덱서블 유형(18)을 생성한다. 코드(10)에서 도시되는 바와 같이, 기민 통신 오퍼레이터(12)는 기민 인덱서블 유형(18)을 DP 콜 사이트(즉, 도 1의 예에서는 forall)로 넘겨주는 데 이용될 수 있다. 이렇게 함으로써, 기민 통신 오퍼레이터(12)는 콜 사이트에 의해 명시

되는 벡터 함수가 모든 컴퓨트 노드들(예를 들어, 도 4에 도시되는 컴퓨트 노드들(121)) 상에 복제되도록 하고, 여기서 각각의 컴퓨트 노드는 컴퓨트 노드에 할당되는 세그먼트들(20)을 수신한다.

[0020] 기민 통신 오퍼레이터(12)는 입력 인덱서블 유형(14)이 세그먼트들(20)로 분해되도록 하고 각각의 세그먼트(20)에는 자원 맵(16)에 의해 명시되는 컴퓨트 노드에 할당된다. 자원 맵(16)은 메모리, 즉 인덱서블 유형(14)이 적어도 하나의 컴퓨트 노드에 걸쳐 저장되는 장소에 대한 세부사항을 제공한다. 자원 맵(16)은 세그먼트들(20)을 명시함으로써 세그먼트들(20)의 모임이 중첩되지 않고 기민 인덱서블 유형(18)을 커버하도록 한다. 자원 맵(16)으로 인해 세그먼트들(20)이 동일하거나 상이한 블록 크기들 및/또는 규칙적인 또는 불규칙한 블록 결합들로 명시되는 것이 가능하다.

[0021] 도 3a 내지 도 3c는 기민 인덱서블 유형(18(1))을 생성하고 이용하는 예들을 도시하는 블록도들이다. 도 3a 내지 도 3c의 예에서, 기민 통신 오퍼레이터(12)는 0에서 23의 번호가 매겨진 요소들을 가지는 6 × 6 매트릭스들(즉, 입력 인덱서블 유형(14(1))을 대응 자원 맵(16)(도 2에 도시됨)에 의해 명시되는 바와 같이 도 3b에 도시되는 기민 인덱서블 유형(18(1))의 9개의 세그먼트들(20))로 분할한다. 각각의 세그먼트(20)는 도 3b에 상이한 음영으로 표현된다. 예를 들어, 제 1 세그먼트(20(1))는 요소들(0, 1, 6 및 7)을 포함하고, 제 2 세그먼트(20(2))는 2, 3, 8 및 9를 포함하고, 기타 등등이다. 기민 통신 오퍼레이터(12)는 또한 세그먼트들(20(1) 내지 20(9))가 하나 이상의 컴퓨트 노드들(121(1) 내지 121(Q))의 세트로 할당되도록 하며, 여기서 Q는 자원 맵(16)에서의 프로토콜에 의해 명시되고 도 3c에서의 화살표(30)에 의해 표시되는 바와 같이, 1보다 더 크거나 1과 같은 정수이다.

[0022] 자원 맵(18)은 블록 분해, 주기 분해, 블록-블록 분해 또는 블록-주기 분해와 같은 임의의 적절한 할당 프로토콜을 통합할 수 있다. 다음의 프로토콜 예들은 세 개의 컴퓨트 노드들(121(1) 내지 121(3))(즉, Q = 3) 또는 4개의 컴퓨트 노드들(121(1) 내지 121(4))(즉, Q = 4)이 있고 세그먼트들(20)이 제 1(즉, 최상의) 행으로부터 시작해서 좌측으로부터 우측으로 행들에 걸쳐 번호가 부여된다(20(1) 내지 20(9))고 가정한다.

[0023] 행 블록 분해 및 Q = 3에 있어서, 입력 인덱서블 유형(14(1))의 36개의 요소들이 3으로 나누어짐으로써 각각의 컴퓨트 노드(121)에는 12개의 요소들이 할당된다. 그러므로, 자원 맵(18)은 요소들(0 내지 11)(즉, 세그먼트들(20(1) 내지 20(3))이 컴퓨트 노드(121(1))로 할당되도록 하고, 요소들(12 내지 23)(즉, 세그먼트들(20(4) 내지 20(6))이 컴퓨트 노드(121(2))로 할당되도록 하며, 요소들(24 내지 35)(즉, 요소들(20(7) 내지 20(9))이 컴퓨트 노드(121(3))로 할당되도록 한다.

[0024] 행 블록 분해 및 Q = 4에 있어서, 입력 인덱서블 유형(14(1))의 36개의 요소들이 4로 나누어짐으로써 각각의 컴퓨트 노드(121)에는 9개의 요소들이 할당된다. 따라서, 자원 맵(18)은 요소들(0 내지 8)이 컴퓨트 노드(121(1))에 할당되도록 하고, 요소들(9 내지 17)이 컴퓨트 노드(121(2))에 할당되도록 하며, 요소들(18 내지 26)이 컴퓨트 노드(121(3))에 할당되도록 하며, 요소들(27 내지 36)이 컴퓨트 노드(121(4))에 할당되도록 한다.

[0025] 열 블록 분해 및 Q = 3에 있어서, 자원 맵(18)은 제 1 및 제 2 행들의 세그먼트들(20)(즉, 세그먼트들 20(1), 20(4) 및 20(7))이 컴퓨트 노드(121(1))에 할당되도록 하고, 제 3 및 제 4 열들의 세그먼트들(20)(즉, 세그먼트들 20(2), 20(5) 및 20(8))이 컴퓨트 노드(121(2))에 할당되도록 하며, 제 5 및 제 6 열들의 세그먼트들(20)(즉, 세그먼트들 20(3), 20(6) 및 20(9))이 컴퓨트 노드(121(3))에 할당되도록 한다.

[0026] 행 주기 분해 및 Q = 3에 있어서, 자원 맵(18)은 요소들(3*k)(k=0 내지 11)이 컴퓨트 노드(121(1))에 할당되도록 하고, 요소들(3*k+1)이 컴퓨트 노드(121(2))에 할당되도록 하고, 요소들(3*k+2)이 컴퓨트 노드(121(3))에 할당되도록 한다.

[0027] 행 주기 분해 및 Q = 4에 있어서, 자원 맵(18)은 요소들(4*k)(k=0 내지 8)이 컴퓨트 노드(121(1))에 할당되도록 하고, 요소들(4*k+1)이 컴퓨트 노드(121(2))에 할당되도록 하고, 요소들(4*k+2)이 컴퓨트 노드(121(3))에 할당되도록 하며, 요소들(4*k+3)이 컴퓨트 노드(121(4))에 할당되도록 한다.

[0028] 행 블록-주기 분해 및 Q = 3에 있어서, 분해는 도 3b에 도시되는 세그먼트들(20(1) 내지 20(9))에 대한 주기 분해이다. 따라서, 자원 맵(18)은 세그먼트들(20(1), 20(4) 및 20(7))이 컴퓨트 노드(121(1))에 할당되도록 하고, 세그먼트들(20(2), 20(5) 및 20(8))이 컴퓨트 노드(121(2))에 할당되도록 하며, 세그먼트들(20(3), 20(6) 및 20(9))이 컴퓨트 노드(121(3))에 할당되도록 한다.

[0029] 행 블록-주기 분해 및 Q = 4에 있어서, 자원 맵(18)은 세그먼트들(20(1), 20(5) 및 20(9))이 컴퓨트 노드(121(1))에 할당되도록 하고, 세그먼트들(20(2) 및 20(6))이 컴퓨트 노드(121(2))에 할당되도록 하며, 세그먼트들(20(3) 및 20(7))이 컴퓨트 노드(121(3))에 할당되도록 하며, 세그먼트들(20(4) 및 20(8))이 컴퓨트 노드

(121(4))에 할당되도록 한다.

[0030] 행 블록-블록 분해 및 Q = 3에 있어서, 자원 맵(18)은 세그먼트들(20(1) 내지 20(3))이 컴퓨트 노드(121(1))에 할당되도록 하고, 세그먼트들(20(4) 내지 20(6))이 컴퓨트 노드(121(2))에 할당되도록 하며, 세그먼트들(20(7) 내지 20(9))이 컴퓨트 노드(121(3))에 할당되도록 한다.

[0031] 자원 맵(16)에 대한 행 및 열 분해 결정은 메모리 레이아웃에 좌우될 수 있다. 예를 들어, 열-우선(column-major) 레이아웃은 적절한 프로토콜을 사용하는 행 분해를 내포한다.

[0032] 하나의 실시예에서, 자원 맵(16)은 자원 세그먼트들의 집합을 포함하고 여기서 각각의 자원 세그먼트는 세그먼트(20)를 자원 뷰(즉, 컴퓨터 노드의 추상화(abstraction))(도시되지 않음)와 연관시킨다. 예를 들어, 인덱서블 유형(14)의 경우:

[0033] grid<rank> parent_grid

[0034] 에 의해 규정되고,

[0035] 여기서 grid<rank>는 두 데이터 부재들을 포함한다:

[0036] extent<rank> M extent;

[0037] index<rank> M offset;

[0038] 예를 들어, 도 3b에서의 제 2 세그먼트(20(2))에서, 형상 및 그리드는 _M_extent= {2,2} 및 _M_offset= {0,1}이고, 제 6 세그먼트(20(6))는 _M_extent={2,2} 및 _M_offset={1,2}를 가진다. 따라서, parent_grid는 다음을 이용하여 분해될 수 있다:

[0039] grid<rank> algorithmic_blocks[M1];

[0040] grid<rank> memory_blocks[M2];

[0041] grid<rank> compute_nodes[M3];

[0042] 여기서, M1, M2, M3 > 0 및 M1 >= M2 >= M3이다. 전형적으로, M3는 M2를 나누고 M2는 M1을 나눈다. algorithmic_blocks, memory_blocks 및 compute_nodes의 셋 모두는 중첩되지 않고 parent_grid를 커버한다. algorithmic_blocks은 구현되고 있는 알고리즘에서 이용되는 분해를 표현하고, memory_blocks는 필요할 때, 노드들 사이에서 메모리가 이동하는 조밀도(granularity)를 표현한다. Compute nodes는 컴퓨터 노드들이 대응하는 데이터를 저장하도록 할당되는 조밀도를 표현한다.

[0043] 모든 algorithmic_block 또는 memory_block이 컴퓨트 노드 상에 자신이 저장되는 장소를 검색할 수 있도록, 그리고 각각의 algorithmic_block이 메모리 블록 상에 자신이 저장되는 장소를 검색할 수 있도록 연관성들이 있다고 가정된다. resource_map이라 칭해지는 클래스는 자 그리드(child grid) 및 resource_view 사이의 연관성을 형성하는 resource_segment들로 생성될 수 있다:

```
template <int rank>
struct resource_segment {
   int _M_id;
   grid<rank> _M_child; // representing compute_nodes[k] for some k
   resource_view _M_resource_view;
};
```

[0044]

[0045] 기민 통신 오퍼레이터(12)를 이용하면, 기민 인덱서블 유형(18)의 데이터는 데이터가 현재 상주하고 있는 컴퓨트 노드를 이용자가 인지하고 있지 않아도 중단 없이 액세스될 수 있다. 형상 parent_grid을 가지는 예시 인덱서블 유형(14)(A)의 경우, A의 저장은 resource_segment의 인스턴스들에 의해 결정된다. index<rank> Index:

[0046] 에서 A의 요소에 액세스하기 위해;

[0047] _Index를 포함하는 자-그리드가 먼저 발견되고 나서, 오프셋:

[0048] index<rank> Offset;

[0049] 이 결정됨으로써

[0050] _Index = child-grid-offset + Offset이 된다.

[0051] 자원 세그먼트 표기법에 있어서, 상기 관계는:

[0052] _Index = _Resource_segment._M child._M_offset +_Offset이다.

[0053] 검색들의 속도를 증가시키기 위해, 다음의 체크는 _Index(_Index가 변함에 따라)가 여전히 _Resource_segment._M _child에 속하는지를 결정하도록 수행된다:

```
index<rank> _Local_offset = _Index - _Resource_segment._M_child._M_offset;
extent<rank> _Local_bounds = _Resource_segment._M_child._M_extent;

if ( _Local_offset < _Local_bounds ) {
        // access data wrt machine _M_resource_view
   ... local_array[ _Local_offset ] ...
}
```

[0054]

[0055] 제공되는 _Index가 속하는 자 그리드 또는 _Resource_segment의 결정은 분해 패턴에 좌우된다. 최악의 경우에 모든 차원에서의 이진 탐색이 이용될 수 있으나 회피되지 않을 수 있다. 모든 타일들이 동일한 범위를 가지는 2048 × 2048 타일 분해에 있어서,

[0056] index<2> _Tile(2048,2048);

[0057] (_Index + Tile - 1) / _Tile

[0058] 과 동일한 _M_child._M_offset을 가지는 _Resource_segment를 찾는다.

[0059] 상기 _Resource_segment(즉, 현재의 resource_segment)는:

[0060] if (_Local_offset <_Local_bounds) { ... }

[0061] 이 위반될 때까지 이용될 수 있고, 이 경우에 새 _Index를 by_Tile로 다시 나누고 반복한다. 이 메커니즘은 새 포함 resource_segment들이 단지 가끔씩 발견될 필요가 있는 장소를 가지는 알고리즘들에 최적일 수 있다.

[0062] 후술되는 국부적-전역-뷰 표현에서, 이용자 인덱스 오퍼레이터들은 if-check(본원에서 바운드 체킹(bounds checking)으로 칭해진다):

[0063] if (_Local_offset <_Local_bounds) { ... }

[0064] 를 생략할 수 있는데, 왜냐하면 이용되는 매 액세스마다 바운드들 내에 있음이 신뢰되기 때문이다. 제공된 자원 세그먼트가 소비되고 다른 자원 세그먼트가 이용될 경우, 이용자는 현재 자원 세그먼트를 리셋하는 기능을 호출하는 데 신뢰될 수 있다. 국지적-전역 뷰의 가장 단순한 형태에서, 모든 세 분해들:

[0065] grid<rank> algorithmic_blocks[M1];

[0066] grid<rank> memory_blocks[M2];

[0067] grid<rank> compute_nodes[M3];

[0068] 은 동일한 크기의 블록들 또는 타일들로 규칙적이다. 인덱서블 유형을 타일들로 분할하는 타일 통신 오퍼레이터가 제 1 분해에 적용되어:

[0069] algorithmic_tiles

[0070] 를 산출한다.

[0071] 개별 타일은:

[0072] algorithmic_tiles(_tile_index)이다.

[0073] 소유자 카피가:

[0074] algorithmic_tiles(_tile_index)

[0075] 에서 개시되면,

[0076] 포함하는 memory_blocks[k1] 및 compute_nodes[k2]가 결정된다. 소유자 compute_nodes[k3]가 다음에 결정되고, 그 후에 memory_blocks[k1]는 compute_nodes[k2]로부터 compute_nodes[k3]로 이동된다.

[0077] 자동화 메모리 이동 조밀도는 흔히 세그먼트들(20)의 서브-그리드 분해보다 조밀도가 더 세밀하다. 예를 들어, 도 3a의 매트릭스가 6144 × 6144 요소 매트릭스를 나타낸다고, 즉, 각각 숫자가 매겨진 알고리즘 블록은 1024 × 1024 데이터 요소들을 나타낸다고 가정하자. 6144 x 6144 매트릭스가 도 3b에서와 같이, 2048 × 2048 컴퓨트 노드들로 분해되어 있다고 가정하자. 게다가, Q=4 및 컴퓨트 노드들(121(1), 121(2), 121(3) 및 121(4))이 블록-주기 분해에 따라 2048 × 2048 블록들(즉, 세그먼트들(20(1) 내지 20(9))에 할당된다고 가정하자. 세그먼트들(20(1), 20(5) 및 20(9))은 컴퓨트 노드(121(1))에 할당되고, 세그먼트들(20(2) 및 20(6))은 컴퓨트 노드(121(2))에 할당되고, 세그먼트들(20(3) 및 20(7))은 컴퓨트 노드(121(3))에 할당되며, 세그먼트들(20(4) 및 20(8))은 컴퓨트 노드(121(4))에 할당된다. 메모리는 이 예에서 1024 × 1024 블록들로 이동될 수 있다. 따라서, 계산이 1024 × 1024 블록으로부터 단일 데이터 요소를 이동시키려고 하면, 전체 1024 × 1024 블록이 이동된다.

[0078] 기민 통신 오퍼레이터(12)로 인해 컴퓨트 노드들 사이에서의 세그먼트들(20)의 이동을 관리하기 위해 데이터 병렬(DP) 알고리즘들이 기민 인덱서블 유형(18)의 세그먼트들(20)의 전체 전역-뷰 표현 또는 국지적-전역-뷰 표현으로 코딩되는 것이 가능하다.

[0079] 전체 전역-뷰 표현으로 인해 DP 알고리즘들은 장면들 뒤에서 자동 소유자-카피 메모리 이동이 발생하는 상태로 단일 컴퓨트 노드 상에서 작동될 것처럼 코딩되는 것이 가능하다. 매트릭스 추가에 대한 예로서, A, B 및 C가 각각 도 3a에 도시된 바와 같이 6144 × 6144 매트릭스들이며, 여기서 각각 숫자가 매겨진 블록은 1024 × 1024 데이터 요소들을 나타낸다고 가정하자. A 및 B는 유효 데이터를 지니지만, C는 할당되지 않지만 반드시 임의의 데이터를 지니는 것은 아니다. 게다가 A, B 및 C는 각각 컴퓨터 노드들(121(1)- 121(Q)) 상에 할당되고, 여기서 Q는 이 경우에 4와 같고 A, B 및 C의 각각의 세그먼트들(20(1) 내지 20(9))은 컴퓨트 노드들(121(1) 내지 121(Q)) 상에 각각 저장된다. 다음의 계산에 있어서:

[0080] C = A + B: 여기서 C(i,j) = A(i,j) + B(i,j): 0 <= i,j < 6

[0081] 각각의 (i,j)는 1024 × 1024 요소들을 나타낸다.

[0082] 소유자 카피는 데이터가 필요한 경우 응답, 즉 C가 계산되고 있는 컴퓨트 노드(121)로 이동되는 것을 의미한다. 이 예에서, A 및 B의 블록들은 C의 대응하는 블록들이 계산 지시들로서 저장되는 컴퓨트 노드들(121)로 이동된다. 그러나, 간단한 매트릭스 추가의 경우, 어떠한 이동도 필요하지 않는데 왜냐하면 A 및 B의 블록들은 동일한 컴퓨트 노드들(121) 상에 C의 대응하는 블록들로서 저장되기 때문이다. 상기 계산;

[0083] C(1,2) = A(1,2) + B(1,2)

[0084] 은 A, B 및 C의 각각에 대해 도 3b에서의 블록(8)을 이용한다. 블록(8)은 A, B 및 C의 각각에 대해 컴퓨트 노드(121(2))에 저장되는 세그먼트(20(2))의 일부이므로, 데이터 이동이 발생하지 않는다. 유사하게, 다음의 계산들은 대응하는 세그먼트들(20) 및 컴퓨트 노드들(121) 상에 발생한다:

[0085] 세그먼트(20(1)), 컴퓨트 노드(121(1))에 대해 C(0,0) = A(0,0) + B(0,0)

[0086] 세그먼트(20(1)), 컴퓨트 노드(121(1))에 대해 C(1,0) = A(1,0) + B(1,0)

[0087] 세그먼트(20(4)), 컴퓨트 노드(121(4))에 대해 C(2,0) = A(2,0) + B(2,0)

[0088] 세그먼트(20(4)), 컴퓨트 노드(121(4))에 대해 C(3,0) = A(3,0) + B(3,0)

[0089] 세그먼트(20(7)), 컴퓨트 노드(121(3))에 대해 C(4,0) = A(4,0) + B(4,0)

[0090] 세그먼트(20(7)), 컴퓨트 노드(121(3))에 대해 C(5,0) = A(5,0) + B(5,0)

[0091] 여기서 세그먼트는 분해의 한 요소를 칭한다:

[0092] grid<2> compute_nodes[9].

[0093] 실제로:

[0094] grid<2> algorithmic_blocks[36];

[0095] grid<2> memory_blocks[18];

[0096] grid<2> compute_nodes[9];

[0097] 여기서 알고리즘 블록들은 규모가 1024 × 1024이고, 메모리 블록들은 규모가 2048 x 1024이며, 컴퓨트 노드들은 규모가 2048 × 2048이다. 그러므로, 매트릭스 추가는 상당히 기초적인 예이다.

[0098] 상기 가정들에 대한 다른 예에서, B의 전치는 다음과 같이 C를 생성하기 위해 A에 추가된다:

[0099] $C = A + B^T$: 여기서 $C(i,j) = A(i,j) + B(j,i)^T$: 0 <= i,j < 6

[0100] 여기서 각각의 (i,j)는 1024 × 1024 요소들을 나타내고 $B(j,i)^T$는 기저의 1024 × 1024 블록의 전치이다.

[0101] 이 경우에, B(j, i)는 C(i,j) (및 A(i,j))이 세그먼트들(20(1), 20(5) 및 20(9))에서의 블록들을 제외한 모든 블록들에 대해 저장되는 컴퓨트 노드(121)로 넘어간다. 예를 들어, 세그먼트(20(1))의 블록들은 C의 세그먼트(20(1))의 블록들에 대한 계산이 다음과 같기 때문에 이동될 필요는 없다:

[0102] $C(0,0) = A(0,0) + B(0,0)^T$

[0103] $C(0,1) = A(0,1) + B(1,0)^T$ // $B(1,0)^T$는 B의 세그먼트(20(1)) 내에 있다.

[0104] $C(1,0) = A(1,0) + B(0,1)^T$ // $B(0,1)^T$는 B의 세그먼트(20(1)) 내에 있다.

[0105] $C(1,1) = A(1,1) + B(1,1)^T$

[0106] 그러나, C의 세그먼트(20(4))의 블록들의 경우:

[0107] $C(2,0) = A(2,0) + B(0,2)^T$

[0108] $C(2,1) = A(2,1) + B(1,2)^T$

[0109] $C(3,0) = A(3,0) + B(0,3)^T$

[0110] $C(3,1) = A(3,1) + B(1,3)^T$

[0111] B 블록들은 컴퓨트 노드(121(2)) 상에 저장되는 세그먼트(20(2))의 블록들로부터 기인하고 C 블록들은 컴퓨트 노드(121(4))에 저장되는 세그먼트(20(4))의 블록들로부터 기인한다. 따라서, B의 블록(2)(즉, $B(0,2)^T$)의 1024 × 1024 요소들은 컴퓨트 노드(121(4))로 이동되고, A(2,0)에 추가되고, C(2,0)에 할당되고, B의 블록(8)(즉, $B(1,2)^T$)의 1024 × 1024 요소들은 컴퓨트 노드(121(4))로 이동되고, A(2,1)에 추가되고, C(2,1)에 할당되고, B의 블록(3)(즉, $B(0,3)^T$)의 1024 × 1024 요소들은 컴퓨트 노드(121(4))로 이동되고, A(3,0)에 추가되고, C(3,0)에 할당되고, B의 블록(9)(즉, $B(1,3)^T$)의 1024 × 1024 요소들은 컴퓨트 노드(121(4))로 이동되고, A(3,1)에 추가되고, C(3,1)에 할당된다.

[0112] 전체 전역-뷰 표현에 있어서, 메모리 이동들은 자동으로 행해지는데 왜냐하면 각각의 블록은 컴퓨트 노드(121)가 블록을 저장하는 정보를 지니기 때문이다. 계산들은 도 4에 도시되고 추가하여 상세하게 후술되는 호스트(101)와 같은, 컴퓨트 노드들(121) 또는 호스트 중 어느 것으로부터도 지시를 받을 수 있다.

[0113] 상기 예의 다른 변형들에서, 다수의 세그먼트들(20)은 동일한 컴퓨트 노드(121)에 할당될 수 있으며, 여기서 컴퓨트 노드들(121)의 수는 세그먼트들(20)의 수보다 더 적다. 게다가, 컴퓨트 노드들(121)의 프로세싱 케이퍼빌리티들은 가중될 수 있어서, 더 빠른 컴퓨트 노드들(121)에는 보다 느린 컴퓨트 노드들(121)보다 더 많은 세그먼트들(20)이 할당될 수 있다. 할당들은 상술한 하나 이상의 프로토콜들에 따라 수행될 수 있다.

[0114] 워크-스틸링(work-stealing)을 이용한 자동 부하 균형은 또한 상기의 변형들에서 구현될 수 있다. 컴퓨트 노드(121)가 자신의 계산들을 완료하면, 컴퓨트 노드(121)는 다른 노드들(121)에 할당되는 계산들을 스틸(steal)하

는 시도를 행한다. 계산들을 지시하는 컴퓨트 노드(121), 또는 가능하다면 호스트는 워크-아이템들의 워크-스틸링 큐(queue)들을 저장하고 여기서 상기 큐들은 소유자 행렬(예를 들어 C) 상의 메모리-이동-조밀도(예를 들어, 1024 × 1024)의 계산을 나타내는 임무들을 담고 있다.

[0115] 4개의 동일 가중 컴퓨트 노드들(121(1) 내지 121(4)) 및 블록-주기 분해 프로토콜을 가진 B 전치 예를 가지는 상기 매트릭스 추가물로부터의 A, B, 및 C에 의해, 다음의 4개의 워크-스틸링 큐들은 다음과 같이 저장될 수 있다.

[0116] 그러므로 상기 도면 및 $C = A + B^T$ 및 메모리-이동-조밀도=1204 × 1204 및 동일하게 가중된(w0=w1=w2=w3=1) 4 머신들 및 블록-주기 분해에 의해:

[0117] queue0는 12개의 임무들로 구성된다 - 각각 세그먼트들(20(1), 20(5), 및 20(9))의 4개의 1024 × 1024 블록들;

[0118] queue1은 8개의 임무들로 구성된다 - 각각 세그먼트들(20(2), 및 20(6))의 4개의 1024 × 1024 블록들;

[0119] queue2는 8개의 임무들로 구성된다 - 각각 세그먼트들(20(3), 및 20(7))의 4개의 1024 × 1024 블록들;

[0120] queue3은 8개의 임무들로 구성된다 - 각각 세그먼트들(20(4), 및 20(8))의 4개의 1024 × 1024 블록들.

[0121] 예를 들어, queue2는 임무들을 포함한다:

[0122] $C(0,4) = A(0,4) + B(4,0)^T$

[0123] $C(0,5) = A(0,5) + B(5,0)^T$

[0124] $C(1,4) = A(1,4) + B(4,1)^T$

[0125] $C(1,5) = A(1,5) + B(5,1)^T$

[0126] $C(4,0) = A(4,0) + B(0,4)^T$

[0127] $C(4,1) = A(4,1) + B(1,4)^T$

[0128] $C(5,0) = A(5,0) + B(0,5)^T$

[0129] $C(5,1) = A(5,1) + B(1,5)^T$

[0130] 각각의 컴퓨트 노드(121)는 워크-스틸링 큐으로부터의 모든 임무들이 완료될 때까지 자신의 대응하는 워크-스틸링 큐의 상부로부터 임무를 취한다. 컴퓨트 노드(121)의 워크-스틸링 큐가 비워지면, 컴퓨트 노드(121)는 다른 컴퓨터 노드(121)에 대응하는 워크-스틸링 큐의 하부로부터 임무를 스틸링한다. 국지적-전역-뷰는 전형적으로 조밀도의 the algorithmic_blocks 레벨에서 타일 통신 오퍼레이터를 통해 가능하다. 정규 타일 분해를 취하면, 타일 통신 오퍼레이터는:

[0131] algorithmic_tiles

[0132] 을 산출하는데 우선적으로 적용된다.

[0133] 개별 타일은:

[0134] algorithmic_tiles(_tile_index)이다.

[0135] 소유자 카피가:

[0136] algorithmic_tiles(_tile_index)에서 개시되면

[0137] 포함하는 memory_blocks[k1] 및 compute_nodes[k2]가 결정된다. 소유자 compute_nodes[k3]가 다음에 결정되고, 그 후에 memory_blocks[k1]은 compute_nodes[k2]로부터 compute_nodes[k3]로 이동된다. 이것은 모두 algorithmic_tiles(_tile_index)에 액세스하는 레벨에서 행해진다. 알고리즘을 구현할 때, 요소(또는 회귀적으로, 더 세밀한 블록)는:

[0138] algorithmic_tiles(_tile_index)(_local_index)으로서 액세스된다.

[0139] 전체 전역-뷰 표현과는 대조적으로, 국지적-전역-뷰 표현은 메모리 이동이 이용자에 의해 명시적으로 지정되도록 한다. 상기 전체 전역-뷰 표현 예들에서, 메모리 이동 조밀도는 1024 × 1024 블록들이었으므로, 전체 1024 × 1024 블록은 컴퓨트 노드(121)가 상기 블록 내의 단일 요소에 액세스하는 경우 컴퓨트 노드(121)로 이동되었다.

[0140] 일부 계산들에서, 계산의 조밀도는 메모리 이동 조밀도보다 더 세밀하고 국지적-전역-뷰 표현은 모든 메모리 블록이 이동되어야 하는 장소를 이용자가 명시적으로 지시하는 장점을 제공한다. 예를 들어, 전체 전역-뷰 표현 예들에서 메모리 이동 조밀도가 2048 × 2048, 즉 2 블록들 중 어느 하나로부터 요소가 이동될 때마다 2개의 블록들이 이동된다고 가정하자. 그러므로 C = A + B^T이 경우, C의 세그먼트(20(4))의 블록들에 대한 계산은:

[0141] $C(2,0) = A(2,0) + B(0,2)^T$

[0142] $C(2,1) = A(2,1) + B(1,2)^T$

[0143] $C(3,0) = A(3,0) + B(0,3)^T$

[0144] $C(3,1) = A(3,1) + B(1,3)^T$

[0145] 이다.

[0146] 각각의 경우에, B 블록들은 컴퓨트 노드(121(2)) 상에 저장되고 C 및 A 블록들(C는 소유자이다)은 컴퓨트 노드(121(4)) 상에 저장된다. 그러므로, 상기 진술들 중 처음 2개는 컴퓨트 노드(121(4))로 이동될 B의 블록(2)(즉, $B(0,2)^T$)의 임의의 요소를 명시적으로 지시함으로써 수행된다. 2048 × 1024 메모리 조밀도로 인해, B의 블록들(2 및 8)(즉, $B(0,2)^T$ 및 $B(1,2)^T$)은 처음 2개의 진술들의 추가들이 컴퓨트 노드(121(4))에 의해 실행되는 것이 가능하도록 컴퓨트 노드(121(4))로 이동된다. 마찬가지로, 상기 진술들 중 마지막 2개는 컴퓨트 노드(121(4))로 이동될 B의 블록(3)(즉, $B(0.3)^T$)의 임의의 요소를 명시적으로 지시함으로써 수행된다. 2048 × 1024 메모리 조밀도로 인해, B의 블록들(3 및 9)(즉, $B(0.3)^T$ 및 $B(1.3)^T$)은 마지막 2개의 진술들의 추가들이 컴퓨트 노드(121(4))에 의해 수행되는 것이 가능하도록 컴퓨트 노드(121(4))로 이동된다.

[0147] 이 예들이 나타내는 바와 같이, 제공된 메모리 이동 블록 상에서 하나 이상의 알고리즘들에 의해 실행되는 많은 임무들이 존재할 수 있도록, 계산의 조밀도는 컴퓨트 노드 조밀도보다 더 미세할 수 있는 메모리 이동 조밀도보다 더 미세할 수 있다. 블록의 요소를 이동시키는 단일 지시로 인해 블록 상에서 동작하는 임무들이 더 효율적으로 실행되는 것이 가능하다. 이용자 및 구현 이 둘 모두는 상기 지시 알고리즘들이 다른 메모리 이동 블록 상에서 동작하기 시작할 때까지 그리고 상기에서 확인된 바와 같이, 알고리즘 블록 분해에 대응하는 타일 통신 오퍼레이터가 필요할 때 실제로 메모리 이동을 지시할 때까지 메모리가 이동될 필요가 있는지를 확인하기 위한 체크를 생략할 수 있다. 만일 1024 × 1024 타일(예를 들어 블록(3))이 이동되어야 한다면, 2048 × 1024를 포함하는 메모리 이동 블록(예를 들어 메모리 이동 블록(3))은 2048 × 2048을 포함하는 컴퓨트 노드 블록(예를 들어 세그먼트(20(2))으로부터 소유자-카피 결정 2048 × 2048 블록(예를 들어, 세그먼트(20(4))으로 이동된다. 이제 블록(9)이 이동되어야 하는 경우, 대응하는 타일에 액세스하는 것은 자신이 포함하는 메모리-이동 블록을 검색하고 이것이 이미 세그먼트(20(4))로 이동되었으므로 더 이상 이동이 필요하지 않다고 결정할 것이다. 상술한 바와 같은 바운드 체킹은 정확한 메모리 이동이 타일 내에서의 실제 데이터 요소 액세스 이전에 타일 레벨에서 발생했기 때문에 생략될 수 있다. 즉:

[0148] algorithmic_tiles(_tile_index).

[0149] 임의의 필요한 메모리 이동이 생성되면:

[0150] algorithmic_tiles(_tile_index)(_local_index)

[0151] 은 각각의 국지적 인덱스를 체크하는 바운드들 없이 액세스될 수 있다. 예를 들어, 상기 계산에서, 각각의 메모리 이동 블록에 대해 2개의 알고리즘 임무들이 존재한다.

[0152] 실제로, 소유자 메모리(예를 들어 C) 상의 소정의 메모리 블록 상에서 수행되는 모든 계산은 큰 임무로 그룹화

될 수 있고 임무 내의 제 1 진술은 단일 메모리 이동 지시일 수 있다. 지시는 하나의 실시예에서 다음과 같이 임무에 대한 표준 C++ 주석의 형태일 수 있다.

[0153] [[move_memory(C, B)]]

[0154] void kernel(field<2, double>&C, const field<2, double>& A, const field<2, double>& B);

[0155] 이 주석을 이용함으로써 컴파일러는 메모리 이동 및 계산을 최적화하고 인터리빙(interleaving)할 수 있다.

[0156] 다음의 코드는 하나의 실시예에서 기민 통신 오퍼레이터(12)의 하나의 구현의 개요를 제공한다.

```
//
// A resource_view is a handle for a physical compute engine.
// The only hard requirement is a memory hierarchy in which to realize data.
// The fundamental property of resource_view is _Buffer<T>, which is
// essentially a concrete 1-dimensional array of type T.
//
struct resource_view;
//
// Given an allocation of memory with a structure to represent it as a _Rank-dimenional
// array (viz., field, cf. infra), construct a partioning or segmentation of it. The allocation of
memory has
// indices as integral points in a rectangular set with offset, as described by a grid (cf.
infra).
//
template <int _Rank>
struct index {
   // overloaded subscript and ctors
   int m_base[_Rank];
};
template <int _Rank>
struct extent {
   // overloaded function-call, subscript and ctors
   unsigned int _M_base[_Rank];
};
template <int _Rank>
```

[0157]

```
struct grid {
   static_assert(_Rank > 0, "rank must be > 0");
   unsigned int total_extent() const {
      unsigned int total_extent = _M_extent[0];
      for (int i = 1; i < _Rank; ++i)
         total_extent *= _M_extent[i];
      return total_extent;
   }
   // various overloads
   extent<_Rank> _M_extent;
   index<_Rank> _M_offset.
};
//
// Here is the fundamental data type of DPC++.
// Various indexable types inherit from fields or other
// indexable types. Each indexable types just gives a
// different view of its parent (viz., base class) and the data
// does not physically change. The overloaded subscript
// operator represents the different view. For example, the transpose
// indexable type would have subscript operator in _Rank=2:
//
// template <typename _Parent_type>
// struct transpose_range : public _Parent_type {
//    element_type& operator[] (const index<2>& _Index) {
//       _Parent_type& base = *static_cast<_Parent_type*>(this);
//       return (*this)[index<2>(_Index[1], _Index[0]);
//    }
// };
//
template <int _Rank, typename _Element_type>
struct field {
   // basic indexable type characteristics
   const static int rank = _Rank;
   typedef _Element_type element_type;
   //
   // Construct _M_multiplier so that it maps _Rank-dimensional => 1-dimensional.
   //
   field(const extent<_Rank>& _Extent, const resource_view& _Resource_view)
      : _M_grid(_Extent), _M_store(_Buffer<_Element_type>(_Grid.total_extent())) {
      // create a row-major multiplier vector
      _M_multiplier[0] = 1;
      if (_Rank > 1)
         _M_multiplier[1] = _M_grid[0];
      if (_Rank > 2)
         for (int i = 2; i < _Rank; ++i)
            _M_multiplier[i] = _M_grid[i-1]*_M_multiplier[i-1];
   }
   //
   // Construct a subfield from another
   //
```

[0158]

```
    template <typename _Other>
    field(const grid<_Rank>& _Grid, const _Other& _Parent)
        : _M_grid(_Grid), _M_store(_Parent.get_store()),
_M_multiplier(_Parent.get_multiplier()) {
    }
    //
    // Checking that _Index is within _M_grid, is elided for now.
    //
    element_type& operator[] (const index<_Rank>& _Index) {
        return _Buffer[ flatten(_Index) ];
    }
private:
    grid<_Rank> _M_grid;
    extent<_Rank> _M_multiplier;
    _Buffer<_Element_type> m_store;
    //
    // Map _Rank-dimensional => 1-dimensional.
    //
    unsigned int flatten(const index<_Rank>& _Index) {
    unsigned int _Flat_index = _Index[0];
    if (_Rank > 1)
        _Flat_index += _Index[1]*_M_multiplier[1];
    if (_Rank > 2)
        for (int i = 2; i < _Rank; ++i)
            _Flat_index += _Index[i]*_M_multiplier[i];
        return _Flat_index;
    }
};
//
// Given a grid, define grid_segment to represent a single segment in the segmentation.
//
//
// _M_parent is the grid being segmented (viz., partitioned).
// _M_child describes the extent and offset of the owned grid segment which must be
within parent.
// _M_id is an identifier for this grid_segment.
//
template <int _Rank>
struct grid_segment {
    int _M_id;
    grid<_Rank> _M_child;
    grid<_Rank> _M_parent;
};
//
// resource_segment associates a grid_segment with a resource_view
//
template <int _Rank>
struct resource_segment {
    grid_segment<_Rank> _M_segment;
    resource_view _M_resource_view;
```

[0159]

```
};
//
// resource_map is a collection of resource segments covering a parent grid.
// resource_map is not marshaled to a device.
// It is a structure that either completely resides on a single compute node or a host.
// resource_map can be specialized for block, cyclic and block-cyclic decompositions,
// as well as sparse, strided, associative, and unstructured.
//
template <int _Rank>
struct resource_map {
   //
   // Create a resource_map by specifying the split points and parent grid.
   // The ith slot of every split point in _Split_points that is != -1, determines the
   // partition points on the ith axis. Let the number of such points be N(i).
   // Then there are N(0)*N(1)*...*N(_Rank - 1) resource segments.
   //
   //  For example, rank=2,
   // parent_grid._M_extent={36,36},
   // _Split_points[0] = {16},
   // _Split_points[1] = {16}, then there are 4 = (1+1)*(1+1) segments of extent {16, 16}.
   //
   // This specifies the compute_nodes
   // decomposition. There are also methods/ctors that
   // allow the specification of  memory_blocks
   // and algorithmic blocks.
   //
   // The resource_map provides the mechanism for
   // both every data element and every algorithmic
   // tile to determine the containing memory movement
   // block and the containing resource_segment.
   // This is enough for an agile field to determine when
   // to move memory and when to just access for free
   // (local-global-view).  Or for global-view, enough
   // information for every data element to find the
   // containing resource_segment whenever the
   // if-check set forth above (i.e., bounds checking) is violated.
   //

//
// Now create an agile indexable type
//
template <typename _Parent_type>
struct agile_range : public _Parent_type {
   const static int rank = _Parent_type::rank;
   typedef typename _Parent_type::element_type element_type;

   agile_range(const _Parent_type& _Parent, const resource_map<rank>&
_Resource_map)
      : _Parent_type(_Parent), _M_resource_map(_Resource_map) {
   }
```

[0160]

```
private:
   resource_map<rank> _M_resource_map;
};
//
// agile communications operator 12 is the standard way to change resource view, usually
to multiple resource views.
//
template <typename _Parent_type>
agile_range<_Parent_type> agile(const _Parent_type& _Parent, const
resource_map<rank>& _Resource_map) {
   return agile_range<_Parent_type>(_Parent, _Resource_map);
}
```

[0161]

[0162] 도 4는 기민 통신 오퍼레이터(12)를 포함하는 데이터 병렬 코드(10)를 컴파일링하고 수행하도록 구성되는 컴퓨

터 시스템(100)의 실시예를 도시하는 블록도이다.

[0163] 컴퓨터 시스템(100)은 하나 이상의 프로세서 패키지들(도시되지 않음)에 하우징되는 하나 이상의 프로세싱 요소들(PE들)(102) 및 메모리 시스템(104)을 구비하는 호스트(101)를 포함한다. 컴퓨터 시스템(100)은 또한 영 이상의 입력/출력 디바이스들(106), 영 이상의 디스플레이 디바이스들(108), 영 이상의 주변 장치들(110), 및 영 이상의 네트워크 디바이스들(112)을 포함한다. 컴퓨터 시스템(100)은 하나 이상의 DP 최적 컴퓨트 노드들(121)을 구비하는 컴퓨트 엔진(120)을 더 포함하고 여기서 각각의 DP 최적 컴퓨트 노드(121)는 하나 이상의 프로세싱 요소들(PE들)(122)의 세트 및 DP 실행파일(138)을 저장하는 메모리(124)를 포함한다.

[0164] 호스트(101), 입력/출력 디바이스들(106), 디스플레이 디바이스들(108), 주변 장치들(110), 네트워크 디바이스들(112), 및 컴퓨트 엔진(120)은 제어기들, 버스들, 인터페이스들, 및/또는 다른 유선 또는 무선 접속들을 포함하는 임의의 적절한 유형, 수, 및 구성을 포함하는 상호접속부들(114)의 세트를 이용하여 통신한다.

[0165] 컴퓨터 시스템(100)은 범용 또는 특정 목적을 위해 구성되는 임의의 적절한 프로세싱 디바이스를 나타낸다. 컴퓨터 시스템(100)의 예들은 서버, 개인용 컴퓨터, 랩탑 컴퓨터, 태블릿 컴퓨터, 스마트폰, 개인용 디지털 보조 장치(PDA), 모바일 전화, 및 오디오/비디오 디바이스를 포함한다. 컴퓨터 시스템(100)의 구성요소들(즉, 호스트(101), 입력/출력 디바이스들(106), 디스플레이 디바이스들(108), 주변 장치들(110), 네트워크 디바이스들(112), 상호 접속부들(114), 및 컴퓨트 엔진(120))은 공통의 하우징(도시되지 않음) 내에 또는 임의의 적절한 수효의 별개의 하우징들(도시되지 않음)에 포함될 수 있다.

[0166] 프로세싱 요소들(102)은 각각 메모리 시스템(104)에 저장되는 명령들(즉, 소프트웨어)를 실행하도록 구성되는 실행 하드웨어를 형성한다. 각각의 프로세서 패키지 내의 프로세싱 요소들(102)은 동일하거나 상이한 아키텍처들 및/또는 명령 세트들을 가질 수 있다. 예를 들어, 프로세싱 요소들(102)은 순차적 실행 요소들, 슈퍼스칼라 실행 요소들, 및 데이터 병렬 실행 요소들(예를 들어, GPU 실행 요소들)의 임의의 결합을 포함할 수 있다. 각각의 프로세싱 요소(102)는 메모리 시스템(104)에 저장되는 명령들에 액세스하고 상기 명령들을 실행하도록 구성된다. 명령들은 기본 입출력 시스템(basic input output system; BIOS) 또는 펌웨어(도시되지 않음), 운영 시스템(OS)(132), 코드(10), 컴파일러(134), GP 실행파일(136), 및 DP 실행파일(138)을 포함할 수 있다. 각각의 프로세싱 요소(102)는 입력/출력 디바이스들(106), 디스플레이 디바이스들(108), 주변 장치들(110), 네트워크 디바이스들(112), 및/또는 계산 엔진(120)으로부터 수신되는 정보에 따라 또는 상기 정보에 응답하여 명령들을 실행할 수 있다.

[0167] 호스트(101)는 OS(132)를 부팅하고 실행한다. OS(132)는 컴퓨터 시스템(100)의 구성요소들을 관리하기 위해 프로세싱 요소들에 의해 실행 가능한 명령들을 포함하고 프로그램이 구성요소들에 액세스하고 상기 구성요소들을 이용할 수 있게 하는 기능들의 세트를 제공한다. 하나의 실시예에서, OS(132)는 윈도우 운영 시스템이다. 다른 실시예들에서, OS(132)는 컴퓨터 시스템(100)과 함께 이용하는데 적합한 다른 운영 시스템이다.

[0168] 컴퓨터 시스템이 코드(10)를 컴파일링하기 위해 컴파일러(134)를 실행하면, 컴파일러(134)는 하나 이상의 실행파일들 - 예를 들어 하나 이상의 GP 실행파일들(136) 및 하나 이상의 DP 실행파일들(138)을 생성한다. 다른 실시예들에서, 컴파일러(134)는 각각 하나 이상의 DP 실행파일들(138)을 포함하도록 하나 이상의 GP 실행파일들(136)을 생성할 수 있거나 어떠한 GP 실행파일들(136)도 생성하지 않고 하나 이상의 DP 실행파일들(138)을 생성할 수 있다. GP 실행파일들(136) 및/또는 DP 실행파일들(138)은 코드(10)의 모든 또는 선택된 부분들을 컴파일링하기 위해 데이터 병렬 확장들에 의한 컴파일러(134)의 호출에 응답하여 생성된다. 호출은 예를 들어 컴퓨터 시스템(100)의 프로그래머 또는 다른 이용자, 컴퓨터 시스템(100) 내의 다른 코드, 또는 다른 컴퓨터 시스템(도시되지 않음) 내의 다른 코드에 의해 생성될 수 있다.

[0169] GP 실행파일(136)은 하나 이상의 범용 프로세싱 요소들(102)(예를 들어 중앙 처리 장치(CPU)들) 상에서 실행되도록 의도되는 프로그램을 나타낸다. GP 실행파일(136)은 하나 이상의 범용 프로세싱 요소들(102)의 명령 세트로부터의 저레벨 명령들을 포함한다.

[0170] DP 실행파일(138)은 하나 이상의 데이터 병렬(DP) 최적 컴퓨트 노드들(121) 상에서 실행되도록 의도되고 최적화되는 데이터 병렬 프로그램 또는 알고리즘(예를 들어 쉐이더(shader))를 나타낸다. 하나의 실시예에서, DP 실행파일(138)은 DP 최적 컴퓨트 노드(121) 상에서 실행되기 전에 디바이스 구동기(도시되지 않음)를 이용하여 DP 최적 컴퓨트 노드(121)의 명령 세트로부터의 저레벨 명령들로 변환되는 DP 바이트 코드 또는 어떤 다른 중간 표현(IL)을 포함한다. 다른 실시예들에서, DP 실행파일(138)은 하나 이상의 DP 최적 컴퓨트 노드들(121)의 명령 세트로부터의 저레벨 명령들을 포함하고 여기서 저레벨 명령들은 컴파일러(134)에 의해 삽입되었다. 따라서, GP

실행파일(136)은 하나 이상의 범용 프로세서들(예를 들어 CPU들)에 의해 직접적으로 실행될 수 있고 DP 실행파일(138)은 하나 이상의 DP 최적 컴퓨트 노드들(121)에 의해 직접적으로 실행될 수 있거나 DP 최적 컴퓨트 노드(121)의 저레벨 명령들에 변환되는 것에 후속하는 하나 이상의 DP 최적 컴퓨트 노드들(121)에 의해 실행될 수 있다.

[0171] 컴퓨터 시스템(100)은 하나 이상의 프로세싱 요소들(102)을 이용하여 GP 실행파일(136)을 실행할 수 있고, 컴퓨터 시스템(100)은 아래에서 추가적으로 상세하게 기술되는 바와 같이 하나 이상의 PE들(122)을 이용하여 DP 실행파일(138)을 실행할 수 있다.

[0172] 메모리 시스템(104)은 명령들 및 데이터를 저장하도록 구성되는 휘발성 또는 비휘발성 저장 디바이스들의 임의의 적절한 유형, 수, 및 구성을 포함한다. 메모리 시스템(104)의 저장 디바이스들은 OS(132), 코드(10), 컴파일러(134), GP 실행파일(136), 및 DP 실행파일(138)을 포함하는 컴퓨터 실행 가능 명령들(즉, 소프트웨어)를 저장하는 컴퓨터 판독 가능 저장 매체를 나타낸다. 명령들은 본원에서 기술되는 바와 같이 OS(132), 코드(10), 컴파일러(134), GP 실행파일(136), 및 DP 실행파일(138)의 기능들 및 방법들을 수행하기 위해 컴퓨터 시스템(100)에 의해 실행 가능하다. 메모리 시스템(104)은 프로세싱 요소들(102), 입력/출력 디바이스들(106), 디스플레이 디바이스들(108), 주변 장치들(110), 네트워크 디바이스들(112), 및 컴퓨트 엔진(120)로부터 수신되는 명령들 및 데이터를 저장한다. 메모리 시스템(104)은 프로세싱 요소들(102), 입력/출력 디바이스들(106), 디스플레이 디바이스들(108), 주변 장치들(110), 네트워크 디바이스들(112), 및 컴퓨트 엔진(120)에 저장된 명령들 및 데이터를 제공한다. 메모리 시스템(104) 내의 저장 디바이스들의 예들은 하드 디스크 드라이브들, 임의 액세스 메모리(random access memory; RAM), 판독 전용 메모리(read only memory; ROM), 플래시 메모리 드라이브들 및 카드들, 및 CD들 및 DVD들과 같은 자기 및 광 디스크들을 포함한다.

[0173] 입력/출력 디바이스들(106)은 이용자로부터의 명령들 또는 데이터를 컴퓨터 시스템(100)에 입력하고 컴퓨터 시스템(100)으로부터의 명령들 또는 데이터를 이용자에게 출력하도록 구성되는 입력/출력 디바이스들의 임의의 적절한 유형, 수, 및 구성을 포함한다. 입력/출력 디바이스들(106)의 예들은 키보드, 마우스, 터치패드, 터치스크린, 버튼들, 다이얼들, 노브(knob)들, 및 스위치들을 포함한다.

[0174] 디스플레이 디바이스들(108)은 텍스트 및/또는 그래픽 정보를 컴퓨터 시스템(100)의 이용자에게 출력하도록 구성되는 디스플레이 디바이스들의 임의의 적절한 유형, 수, 및 구성을 포함한다. 디스플레이 디바이스들(108)의 예들은 모니터, 디스플레이 스크린, 및 프로젝터를 포함한다.

[0175] 주변 디바이스들(110)은 범용 또는 특정 프로세싱 기능들을 수행하기 위해 컴퓨터 시스템(100) 내에 하나 이상의 다른 구성요소들과 동작하도록 구성 가능한 주변 디바이스들의 임의의 적절한 유형, 수, 및 구성을 포함할 수 있다.

[0176] 네트워크 디바이스들(112)은 컴퓨터 시스템(100)이 하나 이상의 네트워크들(도시되지 않음)에 걸쳐 통신하는 것이 가능하도록 구성되는 네트워크 디바이스들의 임의의 적절한 유형, 수, 및 구성을 포함한다. 네트워크 디바이스들(112)은 정보가 컴퓨터 시스템(100)에 의해 네트워크에 전송되거나 컴퓨터 시스템(100)에 의해 네트워크로부터 수신되는 것이 가능한 임의의 적절한 네트워킹 프로토콜 및/또는 구성에 따라 동작할 수 있다.

[0177] 컴퓨트 엔진(120)은 DP 실행파일(138)을 실행하도록 구성된다. 컴퓨트 엔진(120)은 하나 이상의 컴퓨트 노드들(121)을 포함한다. 각각의 컴퓨트 노드(121)는 메모리 계층을 공유하는 계산 자원들의 집합체이다. 각각의 컴퓨트 노드(121)는 하나 이상의 PE들(122)의 세트 및 DP 실행파일(138)을 저장하는 메모리(124)를 포함한다. PE들(122)은 DP 실행파일(138)을 실행하고 DP 실행파일(138)에 의해 발생되는 결과들을 메모리(124)에 저장한다. 특히, PE들(122)은 기민 통신 오퍼레이터(12)를 입력 인덱서블 유형(14)에 적용하여 도 4에 도시되고 위에 추가적으로 상세하게 기술되는 바와 같은 출력 인덱서블 유형(18)을 생성하기 위해 DP 실행파일(138)을 실행한다.

[0178] 데이터 병렬 컴퓨팅(즉, DP 프로그램들 또는 알고리즘들의 실행)에 최적화된 하드웨어 아키텍처를 구비하는 하나 이상의 계산 자원들을 가지는 컴퓨트 노드(121)는 DP 최적 컴퓨트 노드(121)로서 칭해진다. DP 최적화 컴퓨트 노드(121)의 예들은 PE들(122)의 세트가 하나 이상의 GPU들을 포함하는 노드(121) 및 PE들(122)의 세트가 범용 프로세서 패키지 내의 SIMD 장치들의 세트를 포함하는 노드(121)를 포함한다. 데이터 병렬 컴퓨팅에 대해 최적화되는 하드웨어 아키텍처를 구비하는 임의의 계산 자원들(예를 들어 범용 프로세싱 요소들(102)을 구비하는 프로세서 패키지들)을 가지지 않는 컴퓨트 노드(121)는 비-DP 최적 컴퓨트 노드(121)로 칭해진다. 각각의 컴퓨트 노드(121)에서, 메모리(124)는 메모리 시스템(104)(예를 들어 GPU에 의해 이용되는 GPU 메모리) 또는 메모리 시스템(104)의 일부(예를 들어 범용 프로세서 패키지에서 SIMD 장치들에 의해 이용되는 메모리)과 별개일 수 있

다.

[0179] 호스트(101)는 실행을 위해 컴퓨트 노드(121)에 DP 실행파일(138)을 제공하고 상호접속부들(114)을 이용하여 DP 실행파일(138)에 의해 생성되는 결과들을 수신하도록 구성되는 호스트 컴퓨트 노드를 형성한다. 호스트 컴퓨트 노드는 메모리 계층(즉, 메모리 시스템(104))을 공유하는 범용 계산 자원들(즉, 범용 프로세싱 요소들(102))의 집합체를 포함한다. 호스트 컴퓨트 노드는 대칭 멀티프로세싱 아키텍처(symmetric multiprocessing architecture; SMP)와 함께 구성될 수 있고 또한, 예를 들어 비-균일 메모리 액세스(non-uniform memory access; NUMA) 아키텍처를 이용하여 메모리 시스템(104)의 메모리 지역성(memory locality)을 최대화하도록 구성될 수 있다.

[0180] 호스트 컴퓨트 노드의 OS(132)는 DP 실행 파일(138)이 DP 최적 또는 비-DP 최적 컴퓨트 노드(121)에 의해 실행되도록 하는 DP 콜 사이트를 실행하도록 구성된다. 메모리(124)가 메모리 시스템(104)와 분리되어 있는 실시예들에서, 호스트 컴퓨트 노드는 DP 실행파일(138) 및 하나 이상의 인덱서블 유형들(14)이 메모리 시스템(104)으로부터 메모리(124)로 카피되도록 한다. 메모리 시스템(104)이 메모리(124)를 포함하는 실시예들에서, 호스트 컴퓨트 노드는 DP 실행파일(138) 및/또는 메모리 시스템(104) 내의 하나 이상의 인덱서블 유형들(14)의 카피를 메모리(124)로서 지정할 수 있고/있거나 DP 실행파일(138) 및/또는 하나 이상의 인덱서블 유형들(14)을 메모리 시스템(104)의 한 부분으로부터 메모리(124)를 형성하는 메모리 시스템(104)의 다른 부분으로 카피할 수 있다. 컴퓨트 노드(121) 및 호스트 컴퓨트 노드 사이의 카피 프로세스는 비동기로 지정되지 않는 한 동기일 수 있다.

[0181] 호스트 컴퓨트 노드 및 각각의 컴퓨트 노드(121)는 서로에 관계 없이 동시에 코드를 실행할 수 있다. 호스트 컴퓨트 노드 및 각각의 컴퓨트 노드(121)는 노드 계산들을 조정하기 위해 동기화 지점들에서 상호 작용할 수 있다.

[0182] 하나의 실시예에서, 컴퓨트 엔진(120)은 하나 이상의 그래픽 처리 장치(GPU)들이 메모리 시스템(104)과 분리되어 있는 메모리(124) 및 PE들(122)을 포함한다. 이 실시예에서, 그래픽 카드(도시되지 않음)의 구동자는 GPU들의 PE들(122)에 의한 수행을 위해 DP 실행파일(138)의 바이트 코드 또는 일부 다른 중간 표현(IL)을 GPU들의 명령 세트로 변환할 수 있다.

[0183] 다른 실시예에서, 컴퓨트 엔진(120)은 하나 이상의 범용 프로세싱 요소들(102)을 구비하는 프로세서 패키지들 내에 포함되는 하나 이상의 GPU들(즉, PE들(122)) 및 메모리(124)를 포함하는 메모리 시스템(104)의 일부의 결합으로부터 형성된다. 이 실시예에서, 바이트 코드 또는 DP 실행파일(138)의 일부 다른 중간 표현(IL)을 프로세서 패키지들 내의 GPU들의 명령 세트로 변환하기 위해 컴퓨터 시스템(100) 상에 추가 소프트웨어가 제공될 수 있다.

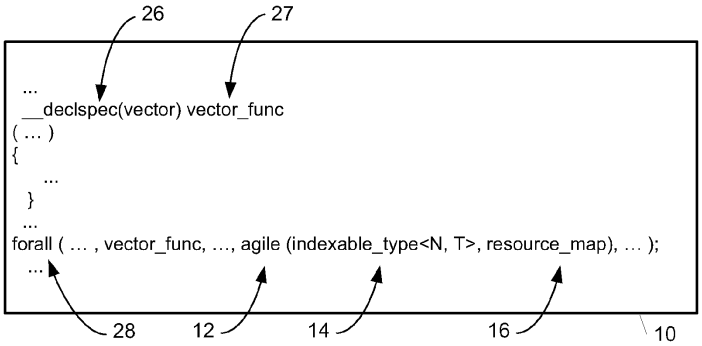
[0184] 추가 실시예에서, 컴퓨트 엔진(120)은 메모리(124)를 포함하는 메모리 시스템(104)의 일부 및 프로세싱 요소들(102)을 포함하는 하나 이상의 프로세서 패키지들 내의 하나 이상의 SIMD 장치들의 결합으로부터 형성된다. 이 실시예에서, 바이트 코드 또는 DP 실행파일(138)의 일부 다른 중간 표현(IL)을 프로세서 패키지들 내의 SIMD 장치들의 명령 세트로 변환하기 위해 컴퓨터 시스템(100) 상에 추가 소프트웨어가 제공될 수 있다.

[0185] 또 다른 실시예에서, 컴퓨트 엔진(120)은 메모리(124)를 포함하는 메모리 시스템(104)의 일부 및 프로세싱 요소들(102)을 포함하는 하나 이상의 프로세서 패키지들 내에 하나 이상의 스칼라 또는 벡터 프로세싱 파이프라인들의 결합으로부터 형성된다. 이 실시예에서, 바이트 코드 또는 DP 실행파일(138)의 일부 다른 중간 표현(IL)을 프로세서 패키지들 내의 스칼라 프로세싱 파이프라인들의 명령 세트로 변환하기 위해 컴퓨터 시스템(100) 상에 추가 소프트웨어가 제공될 수 있다.

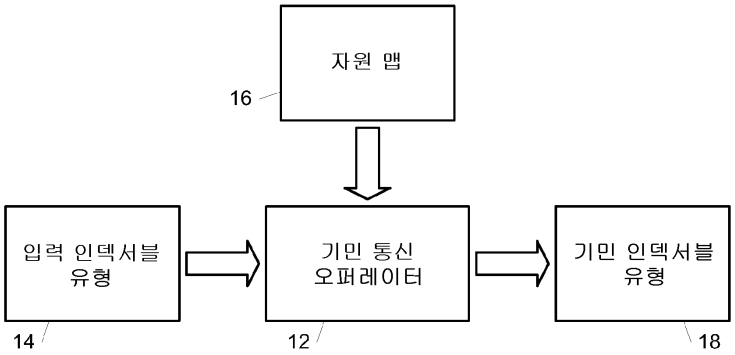
[0186] 본원에서 특정 실시예들이 도시되고 기술되었을지라도 다양한 대체 및/또는 등가의 구현예들이 본 발명의 범위를 벗어나지 않으면서도 도시되고 기술된 상기 특정 실시예들을 대체할 수 있음이 당업자에 의해 인정될 것이다. 본 출원은 본원에서 논의되는 특정 실시예들의 적응들 및 수정들을 포괄하도록 의도된다. 그러므로, 본 발명은 청구항들 및 이의 등가물들에 의해서만 제한되도록 의도된다.

도면

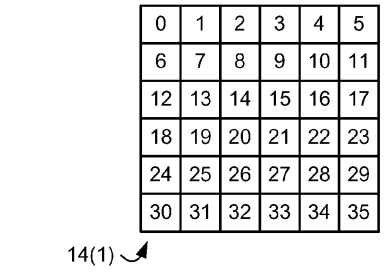
도면1


26
27
...
__declspec(vector) vector_func
(…)
{
...
}
...
forall (… , vector_func, …, agile (indexable_type<N, T>, resource_map), …);
...
28
12
14
16
10

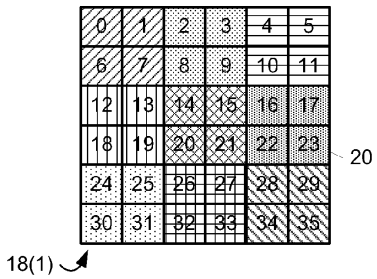
도면2


자원 맵
16
입력 인덱서블 유형
기민 통신 오퍼레이터
기민 인덱서블 유형
14
12
18

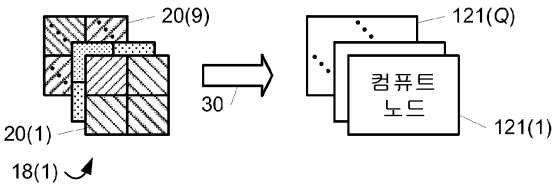
도면3


0 1 2 3 4 5
6 7 8 9 10 11
12 13 14 15 16 17
18 19 20 21 22 23
24 25 26 27 28 29
30 31 32 33 34 35
14(1)

(a)

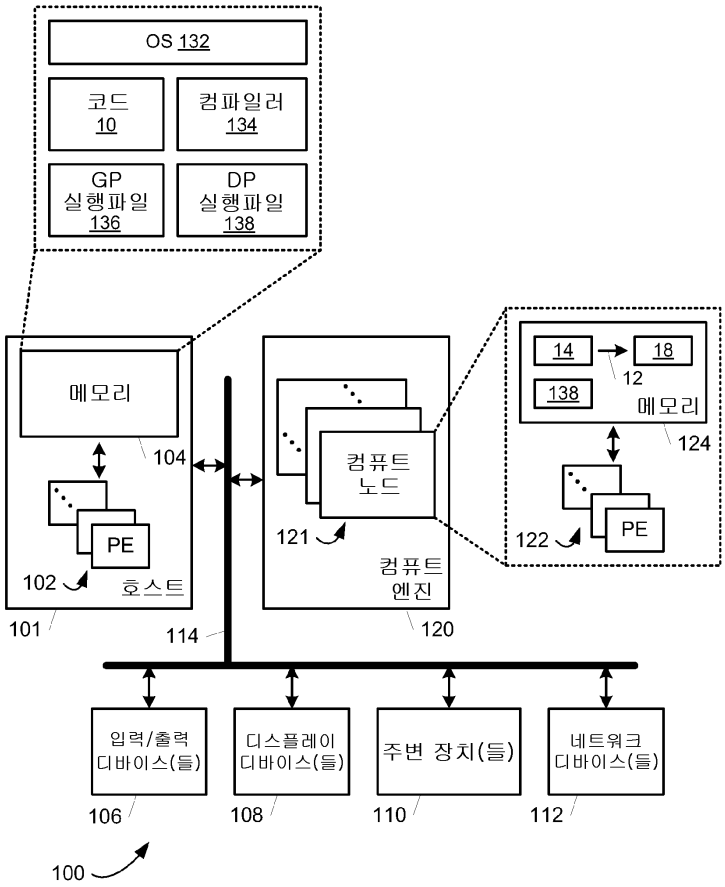

20
18(1)

(b)


20(9)
30
121(Q)
컴퓨트
노드
121(1)
20(1)
18(1)

(c)

도면4


OS 132
코드
10
컴파일러
134
GP
실행파일
136
DP
실행파일
138
메모리
104
PE
102
호스트
101
114
컴퓨트
노드
121
컴퓨트
엔진
120
14
18
12
138
메모리
124
122
PE
입력/출력
디바이스(들)
106
디스플레이
디바이스(들)
108
주변 장치(들)
110
네트워크
디바이스(들)
112
100