2.1 Static and Dynamic Memory Access Behanior of Parallel Applications

**2.2 Thread Mapping Concepts**

병렬 프로그램은 여러 이유로 메모리 액세스 동작을 변경한다 -> Dynamic memory access

Dynamic memory access 는 두가지 유형 (Between executions, During execution) 이러한 사항이 아니라면 Static memory access

Between exectuion

dependent on input data, dependent on number of threads, changing memory access , 응용 프로그램의 지정된 매개변수

During exectution

Algorithmic changes(phase), creation/destructio of threads, allocation/deallocation of memory

Static 경우에는 스레드, 런타임 마이그레이션 수행 할 필요 없음

execution time에서 behavior이 나타나지만, 다음 exectution에서 behavior을 변경하지 않는다면 trace를 사용하여 동작 분석가능

betwenn exectution time에 behavior이 변경된다면 과거 탐지된 behavior이 향후 execution에 유효하지 않기 때문에 새로운 trace를 사용해야한다

2.2.1 Suitable Architectures

CMP, SMT 와 같은 많은 프로세서 아키텍쳐가 존재한다.

이러한 아키텍쳐에서 병렬 어플리케이션의 각 스레드를 실행할 위치를 결정하는 것은

Shared Cache와 Intercommunication을 얼마나 효율적으로 사용하는지에 상당한 영향이 있다.

2.2.2 Baseline Thread to Core Assignment

대부분의 각 코어에 해당 스레드를 할당하는 것은 OS 스케쥴러에 의해 처리됨

현재 Linux에서 사용하는 스케쥴러는 Balance, Fairness에 초점 (Memory Access에 대해 정보를 사용하지 않음)

따라서 대부분의 스레드 맵핑 기법은 명백히 스레드를 코어에 바인딩 시켜 OS 스케쥴러를 재정의 한다.

2.2.3 Policy Goals

스레드 맵핑 기법은 Communication에 대한 Locality, Balance를 높이는 것에 집중

Locality는 Communication이 많은 스레드에 대해 하드웨어 조건에서 가깝게 할당 시키는것을 목표로 한다.

Balance는 Cache와 Intercommunication이 처리하는 통신 량을 공정하게 분배하는것을 목표로 한다.

2.2.4 Policy Types

스레드 매핑 기법은 두가지 방법으로 수행가능

1. Allocation Policy에서는 각 스레드는 OS, Runtime 옵션을 통해 특정 코어에 할당되며 실행이 종료될때 까지 코어에 남아있다.

이 정책은 Runtime에 Overhead를 부과하지 않지만 실행중 동작 변경에 대해 대응할 수없다.

2. Migration Policy는 Communication의 통신량에 따라 Runtime동안 코어간에 스레드가 마이그레이션된다.

이 정책은 Runtime에 Overhead를 부과하며 Migration으로 인한 Cache miss, TLB shootdown도 유발한다.

2.2.5 Determining the Thread Mapping

Thread 매핑 기법을 결정하려면 두가지 정보가 필요

1. 스레드가 Shared Memory에 Access하는 방법을 알아야 한다. (이는 보통 Matrix형태로 표현)

2. Processor, Core, Cache 레벨과 같은 하드웨어 하이라키에 대한 정보가 필요하다.

Graph 분석, 및 하드웨어 Tree 계층 분석으로 많은 연구가 있었다.

2.2.6 Thread Mapping Example

0-4 Thread 가정, SMT architecture 사용

Round-robin Policy => 0 1,, 2 3,, 4

Locality Mapping => 0 4,, 1 3,, 2

Balance Mapping => 0 4,, 1 ,, 3 ,, 2

**2.3 Data Mapping Concepts**

동일 노드에 동일 데이터에 액세스하는 스레드를 배치

스레드 및 NUMA노드에 의해 메모리 페이지가 액세스 되는 방법을 분석, 적합한 노드에 매핑

2.3.1 Underlying Architectures

NUMA가 기존 UMA 시스템보다 병렬 확장성을 개선할 수 있따는 것을 보여줌. ( == Intel QPI, AMD HT)

최근엔 동일한 칩 내에 여러 메모리 컨트롤러가 포함될 수있으며, 단일 프로세서에서도 NUMA동작이 가능

2.3.2 Basline Memory Page to Node Mapping

Basline Data Mapping 형식은 스레드 매핑 Policy와 대조적으로 Static하다.

Home-node Allocation : 스레드가 메모리를 할당할 때 마다 이 메모리는 스레드가 실행 중인 노드에 할당

Interleave Policy : 노드는 virtual page address의 비트에 의해 결정, 연속된 페이지는 서로 다른 노드에 매핑됨,

각 노드의 페이지수가 비슷해지고, 균일하게 Access된다. 대신 Locality가 낮다.

First-touch : 각 페이지는 NUMA node에 매핑, page fault handler를 실행하는 동안 페이지에 대한 첫번째 메모리 엑세스가 수행

이는 페이지에 대한 첫번째 access를 수행하는 스레드가 그 페이지에 대한 후속 접근을 수행할 가능성이 농후 -> 어플리케이션 개발자들의 데이터 매핑에 영향

2.3.3 Policy Goals

스레드 매핑 기법과 같은 Locality, Balance 균형을 향상시키는 초점 목표

자주 Access 하는 page를 local memory에 배치, 하지만 이런 정책은 다른것보다 더많은 Access를 수행하는 상황에서 불균형 초래

Balance정책은 각 컨트롤러가 처리하는 Access수를 균등화 하여 컨트롤러 과부하 방지, 많은 병렬 프로그램에서는 Locality를 개선하면 Balance개선

2.3.4 Policy Types

Allocation, Migration, Replication

Allocation Policy는 NUMA node에 할당되고 실행이 완료될때 까지 해당노드에 유지. Runtime에 Overhead 부과 안하지만 변경 대응 불가

Migration Policy는 메모리 엑세스 동작에 따라 Runtime동안 NUMA node간에 마이그레이션. 이는 기능 자체 호출 Overhead 뿐만아니라

노드간 데이터 복사로 인한 Runtime Overhead 증가. Runtime중 변경에 대응 가능

Replicaton Policy는 여러 NUMA node에서 Access 하는 페이지는 해당 node에서 복제됨. 복제는 Locality, Balance를 동시 향상 이점 가짐

페이지 중복으로 인한 메모리량 감소, 메모리가 수정된다면 다른 복사본으로 변경사항 전파. 따라서 수정되지 않는 데이터에 유용

2.3.5 Determining the data Mapping

Thread 매핑 기법보다 간단.

프로세스 동작은 NUMA node에서 페이지당 메모리 액세스 수를 보여주는 페이지 사용패턴의 형태로 나타냄.

2.3.6 Data Mapping Example

Round-robin Policy => 0 , 1, 2, 3

Allocation Mapping => 2 3 ,1 , 0 ,

Migration Mapping => 2 3, 1 , 0 ,

Replication Mapping => 2 3 , 1, 0 , 3

**2.4 Benefit of Improved Mappings**

병렬 어플리케이션에서 Shared, Private 데이터에 대한 메모리 액세스를 개선하는 것을 목표로 한다.

2.4.1 Performance Improvements

Thread 매핑 기법은 Interconnection 효율을 향상, 트래픽 감소 ,Cache miss 감소, 가용 Cache 공간 증가(동일 공유 캐시 상황)

Data 매핑 기법은 remote 메모리 뱅크에 액세스 수 줄임, 메모리 인접성 증가, interconnection 효율을 향상, 트래픽 감소

2.4.2 Energy Consumption Improvements

어플리케이션 실행시간 단축, Cache miss 및 트래픽 감소시 동적 에너지 소비 감소

실행 시간에 감소하는 비례 소비 에너지 감소

**3. A taxonomy of affinity-based mapping mechanisms**

Affinity-based mapping mechanisms은 메모리 액세스 동작의 Analysis와 매핑 Policy로 구성

중요한 특징은 실행시기(실행전 정보를 사용할수 있는지 여부), 동작을 설명하는데 사용되는 메트릭(HW,OS,Rutime Env), 하드웨어의 수정 가능 여부

분석에 따라 스레드와 데이터를 배치해야하는 정책을 결정(목표(locality,balance), 적용시간(before, during), 환경 수정 등의 특징)

**3.1 User-Level Mechanisms**

for single parallel applications executed

3.1.1 Source Code Changes

소스코드 또는 동작 분석에 의존(사전 실행)

3.1.2 Compiler Analysis

컴파일러 분석에 기초한 메커니즘

컴파일러 또는 해당 런타임 지원 라이브러리에서 매핑을 수행

동적 메모리 액세스 동작을 지원하려면 온라인에서 분석을 수행하는 응용프로그램에 코드를 삽입하여 동적으로 분석할 필요가 있다.

3.1.3 Offline Profiling

1. 어플리케이션은 메모리, 통신 trace를 통해 메모리 액세스 동작을 결정하기 위해 프로파일링. 이후 분석하여 개선된 매핑 결정

2. 어플리케이션 실행중 결정된 매핑이 적용. 매핑은 일반적으로 소스코드 수정 및 런타임 옵션 수정

프로파일링 시간이 길어질수도 있으며, 실행중 동작 변경하는 어플일경우 사용불가, 최소한의 Runtime Overhead만 발생

3.1.4 Online Profiling

실행중에 수집된 정보를 사용하여 매핑을 수행

Cache miss, IPC 같은 동작에 대한 간접 정보를 사용하여 동작을 추정

실행전에 비용이 많은 분석을 수행할 필요가 없음, 대신 정확도와 Overhead 가 중점

메모리 액세스에 대한 정보를 수집하면 보통 정상 실행과 비교해 오버헤드가 상당히 증가

3.1.5 Runtime Options

일부 스레드 및 데이터 매핑은 응용프로그램을 수정하지 않고도 라이브러리 및 커널을 포함한 런타임 환경의 옵션으로 적용 가능

스레드 매핑은 OpenMP, MPI 런타임 환경에 Compact mapping 제공

데이터 매핑 옵션은 응용프로그램에 대한 인터리브 정책을 지정하여 OS에 의해 제공

따라서 소스코드 분석이나 프로파일링과같은 동작의 분석 필요

**3.2. System-Level Mechanisms**

for multiple parallel applications executed

일반적으로 OS나 Hardware에서 구현되므로 Privileges 나 Superuser Access 가 필요하다.

시스템 수준에서 작동하기 때문에 응용프로그램의 데이터 구조에 대한 자세한 정보가 없어 데이터 구조가 아닌 Page 위에서만 데이터 매핑 정책을 결정할 수 있다.

3.2.1 Offline Profiling

매핑 전에 별도의 프로파일링 단계 필요

매핑은 소스코드 수정 및 런타임환경의 옵션 사용

Special hardware counter, hardware-based memory tracers을 사용할 수 있다.

3.2.2 Online Profiling

시스템 수준 통계자료를 사용해 어플리케이션 실행 중 매핑 결정(운영체제 기능을 호출해 매핑 수행)

Page fault, TLB misses, cache coherence message 등이 포함 된다.

**3.3 Characteristics of Taxonomy Types**

3.3.1 Effort for Develop and User

모든 매핑 유형에 대해 소스코드 변경을 제공하면 개발자 및 사용자에게 높은 오버헤드

3.3.2 Availability of Prior Information

Behavior이 실제로 발생하기 전에 메모리 액세스 동작에 대한 정보를 이용

3.3.3 Need for a Previous Execution

실행 간에 Behavior이 변화하는 애플리케이션의 경우, 이전의 실행은 오버헤드를 유발하거나 잘못된 매핑을 적용하는 결과를 초래할 수 있다.

3.3.4 Support for Dynamic Behavior

애플리케이션이 실행 중에 dynamic behavior만 갖는 경우 애플리케이션 시작전이나 초기화전에 정적 매핑만 하는 메커니즘은 제한된 개선만 가져올 수 있다. 런타임에서 변경사항에 반응할 수 있는 메커니즘은 더 높은 런타임 오버헤드 대신 더 나은 Dynamic behavior을 처리할 수 있다.

3.3.5 Change to Application or Runtime Libraries

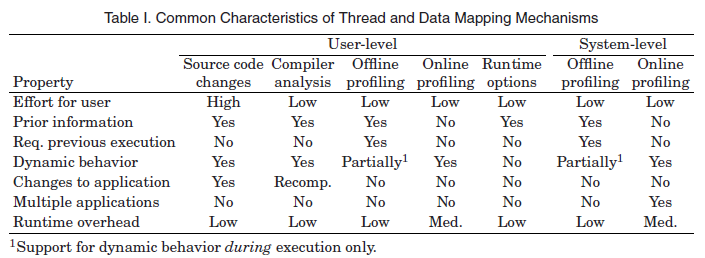
애플리케이션이나 런타임 라이브러리 변경이 필요한 매핑 메커니즘은 더 어렵고 덜 일반적이다.

3.3.6 Support for Multiple Applications

애플리케이션들이 동시에 실행된다면 다른 애플리케이션에 대한 매핑 결정이 서로 방해가 될 수 있다.

3.3.7 Runtime Overhead

런타임 오버헤드는 매핑이 실행 중인 애플리케이션에 미치는 영향을 나타냄. Allocation 정책은 데이터 복사를 포함하지 않기 때문에 일반적으로 Migration, Replication을 기반으로 하는 정책보다 오버헤드가 적다.



**4. Thread Mapping Mechanisms**

Taxonomy and characteristics 에 따른 스레드 매핑 메커니즘에 대한 개요

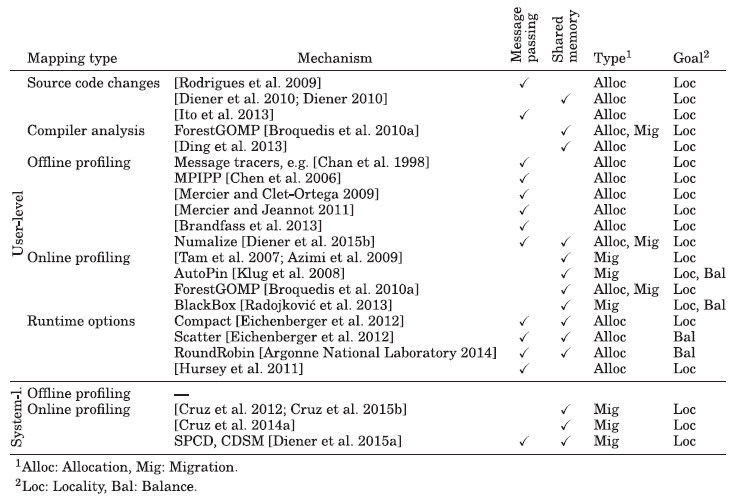
User Level에서는 소스코드 변경을 기반으로 하는 메커니즘이 적다

오프라인 프로파일링 기반 메커니즘이 다수인데 이는 MPI 기반 애플리케이션에서 Message Passing 기능을 비교적 낮은 오버헤드로 계측하여 Message를 trace 가능

User Level 온라인 프로파일링은 Shared Memory 애플리케이션에 더 적합

System Level에서는 온라인 프로파일링 메커니즘이 다수, 하드웨어 및 운영체제에서 수집한 통계를 기반으로 함

스레드 매핑은 보통 단일 애플리케이션을 대상으로 하며 매핑을 수행하기 위해서는 적은 데이터만 필요하기 때문에 User Level에서 구현된다.



**5. Data Mapping Mechanisms**

User Level에서는 소스코드 변경 기반 메커니즘이 다수, 스레드 매핑과 반대인데 이는 사용자 메커니즘을 사용하여 해결하기 어려운 데이터 매핑 복잡성이 높기 때문.

또한 컴파일러 분석은 단순히 메모리 엑세스 패턴을 캡쳐하는 데 종종 제한되어 적고, 오프라인 분석은 오버헤드가 상당이 높아 적다.

온라인 프로파일링은 System Level에서 더 높은 수준으로 구현될 수 있기 때문에 적다.

System Level에서는 하드웨어 카운터로부터 메모리 액세스 동작에 대한 정보를 수집하여 매핑을 결정하는 오프라인 프로파일링 메커니즘이 제안되었다. 온라인 프로파일링에서는 하드웨어 및 운영체제 통계를 수집하여 런타임에 메모리 페이지를 마이그레이션 한다.

스레드 매핑과 데이터 매핑의 큰차이는 온라인 프로파일링에서 나타나는데 이는 User Level에서는 얻을 수 있는 데이터 정보량이 부족하기도 하고 또한 User Level에서 매핑하는게 어렵기도 하기 때문이다.

대부분은 Migration을 기반으로 동작한다.

**6. Thread and Data Mapping Example**

Static Memory Access, Thread 간 많은 양의 Communication, OpenMP 을 가지는 Ondes3D 애플리케이션 커널을 가지고 연구한다.

6.1 Mapping Mechanisms

4가지 유형의 매핑 메커니즘 사용, 모두 스레드와 데이터 매핑을 수행.

6.1.1. Source Code Change

Thread간 통신 패턴을 분석하여 애플리케이션 시작단계에서 스레드가 생성된 후 각 스레드는 system call을 통해 코어에 바인딩되어 통신을 많이하는 Thread가 메모리 계층 구조에 가까운 코어에 배치되고 캐시를 공유한다.

Data mapping은 데이터 구조에 access하는 스레드를 수동으로 결장하고 이 스레드에 의해 데이터 구조 일부를 초기화함으로써 각 스레드에서 Forced-First touch를 통해 구현되었다. 따라서 페이지는 Memory access의 Locality와 Balance를 개선하여 대부분의 액세스를 수행하는 Thread가 할당된 코어가 있는 NUMA node에 위치한다.

6.1.2. Offline Profiling at the User Level

Numalize 메커니즘을 사용하여 Memory Access Trace를 기반으로 오프라인 프로파일링을 수행한다. Numalize는 스레드와 데이터를 지정된 위치에 할당하는 커널 모듈을 통해 애플리케이션에 적용되는 최적화된 스레드 및 데이터 매핑을 결정한다.

6.1.3 Runtime Options

Gcc의 OpenMP 구현에 의해 제공되는 컴팩트 스레드 매핑(GOMP\_CPU\_AFFINITY 환경변수)을 통해 Ondes3D를 실행한다. 데이터 매핑을 통해 Locality와 Balance를 향상 시킬 수 있는 런타임 환경 옵션이 없기 때문에 Numactl을 사용하여 데이터 매핑 정책을 구현하였다.

6.1.4 Online Profiling at the System Level

Linux 커널의 최근 릴리즈의 일부인 NUMA Balancing 기술을 선택하였다. NUMA Balancing은 동일한 데이터에 액세스하는 스레드가 동일한 NUMA node에 있도록 스레드를 마이그레이션한다. 또한 메모리 페이지가 액세스 되는 노드로 마이그레이션 된다.

**6.3 Result**

데이터 매핑으로 인한 200%이상의 큰 성능 향상이 확인된다.

Source Code Change와 Offline Profiling 이 가장 큰 성능향상이 관찰되지만 이는 코드변경 및 재컴파일 시간 소모 또는 Memory trace overhead를 요구함으로 가장 높은 오버헤드를 갖는 메커니즘이기도 하다. 런타임 환경옵션은 데이터 매핑을 통한 Locality 개선이 부족하여 다른 메커니즘만큼 높지는 않았지만 상당한 성능 향상을 야기했다. 애플리케이션 동작에 대한 깊은 분석없이 오버헤드가 적게 성능을 향상시키는 매력적인 방법이 될 수 있다. 온라인 프로파일링은 애플리케이션 동작에 대해 학습해야 하기 때문에 입력 데이터가 큰 집합에 더 적합하다.

