□ 기술해설 □

대규모 병렬처리 컴퓨터의 핵심 기술

서울대학교 한상영*
광운대학교 최영근*
육군사관학교 원영주**

목 차

1. 서 론
2. MPP의 기본 문제 : 지연시간 및 동기화
 2.1 지연시간
 2.2 동기화
3. 공유 메모리 구현 기법
 3.1 메시지 전달을 통한 공유 메모리 구현
 3.2 전역 캐쉬를 사용한 공유 메모리 구현
 3.3 캐쉬 일관성 유지 프로토콜
4. 상호연결망
5. 결 론

1. 서 론

마이크로프로세서 성능의 발달에 힘입어 개인용 컴퓨터와 워크스테이션과 같은 중·소형 컴퓨터의 성능이 획기적으로 발전하고 있음은 주지의 사실이다. 지난 5년간 마이크로프로세서의 성능은 4~6 배에 이르는 향상을 가져왔다. 이러한 변화는 고성능을 요구하는 대형컴퓨터나 또는 빠른 연산을 필요로 하는 수퍼컴퓨터의 분야에도 영향을 미쳐 새로운 전환기를 맞이하고 있다.

기존의 다중프로세서 시스템이 보통 2~16개의 프로세서를 사용하고 있는데 반해 대규모 병렬처리(MPP : Massively Parallel Processing) 컴퓨터는 수백~수천까지 프로세서 개수를 확장할 수 있다. 대규모 병렬처리 컴퓨터의 목적은 빠른 진보를 보이고 있는 고성능 마이크로프로세서를 대량으로 이용하여 기존의 수퍼컴퓨터 또는 대형컴퓨터(mainframe)를 능가하는 성능을 얻는 것이다.

대규모 병렬처리 컴퓨터의 구조로는 크게 SIMD(Single Instruction Multiple Data)와 MIMD(Multiple Instruction Multiple Data)를 들 수 있다. SIMD에서 모든 프로세서들은 동일한 명령어(instruction)를 실행한다. 물론 각 프로세서들에 주어지는 데이타는 서로 다른 것들이다. 프로세서들은 매 사이클마다 서로 다른 데이타를 갖고 동기적으로 명령어를 실행하므로, 범용적인 적용이 어렵다. 특정 범주의 응용 분야, 예를 들어 각 점에 대해 동일한 연산을 수행하는 영상 처리(image processing) 같은 분야에서 SIMD 모델이 월등히 우수하기는 하지만, 이와 같은 응용 범위의 한정성은 MIMD가 범용 컴퓨터로서의 상업성이 보다 좋음을 말해준다. MasPar를 비롯한 몇몇 컴퓨터 회사에서 여전히 SIMD의 개발을 추진해오고 있지만, 본 글에서는 MIMD 모델을 기반으로 한 컴퓨터로 한정하여 다루고자 한다.

일반적인 대형컴퓨터나 수퍼컴퓨터에서 그러하듯이 MIMD 모델의 컴퓨터에서 다수의 프로세서들은 단일 문제를 풀기 위해 사용되거나 또는 여러개의 문제들을 동시에 풀기 위해 사용된다. 현재 일반적으로 많이 사용되는 대칭형 다중프로세서(symmetric multiprocessor) 시스템은 모든 프로세서가 하나의 메모리를 동등한 형태로 공유하고 있는데, 이것은 프로세서

*종신회원
**정 회 원

대규모 확장에 기술적인 한계를 갖는다. 따라서 대부분의 MPP는 분산 메모리 형태를 갖는다. 물리적으로 분산 메모리 형태라고 해도 KSR [1], SCI[2], DASH[3][4] 등과 같이 단일 주소 공간의 공유 메모리 방식을 지원하는 것과 CM-5[5][6], nCube[7] 등과 같이 다중 주소 공간의 메시지 전달 방식을 지원하는 것으로 나눌 수 있다. 공유 메모리를 지원하는 전자의 경우, 메모리 연산은 여러 프로세서 캐쉬에 복사본을 남기게 된다. 따라서 공유 메모리 방식에서는 이들 복사본 사이의 일관성(coherency)을 유지하기 위한 캐쉬 프로토콜 및 메카니즘이 필요하다. 메시지 전달 방식은 각 프로세서 사이의 캐쉬 일관성 문제는 피할 수 있지만, 메시지 형성 및 전달에 따르는 부담이 남는다.

메시지 전달이나 캐쉬 관리는 기본적으로 분산되어 있는 [프로세서-메모리] 노드간의 통신 지연시간(communication latency) 문제를 안고 있다. 통신 지연은 하드웨어의 물리적 분할로부터 기인한다. 과거의 수퍼컴퓨터와 같은 접근 방식에서는 프로세서 개수가 그리 많지 않았기 때문에 이러한 문제가 그다지 중요하지 않았지만, 대규모의 [프로세서-메모리]로 구성되는 MPP 시스템에서는 이의 문제가 매우 심각하다. 노드간의 통신 지연이나 대역폭은 상호연결망의 특성에 좌우되는데, 상호연결망은 링(ring), 메쉬(mesh), 하이퍼큐브(hypercube), 팻트리(fat-tree), 큐브 연결 사이클(cube connected cycle), 다단계 네트워크(multistage network), 크로스바 네트워크(crossbar network) 등과 같이 다양하게 구성될 수 있으며, 이에 대한 연구가 계속 진행되고 있다.

한편, 단일 문제에 대한 병렬 실행은 프로그램을 병렬적으로 실행할 수 있는 단위로의 분할을 필요로 한다. 여러가지 계산 모델에 따라 매우 작은 크기의 소단위(fine-grain) 분할부터 비교적 큰 대단위(coarse-grain) 분할에 이르는 병렬 실행 단위가 있다. 이러한 것들은 어떠한 방식의 모델, 분할 크기 등을 따르든지 공통적으로 동기화(synchronization) 작업을 필요로 한다. 통신 지연이 물리적 분할에 기인했던 것에 반해 동기화는 논리적 분할, 즉 프로그램의 분할에 기인한다. 동기화는 이들 분할된 병렬 실행 단위들의 실행에 있어서 시간적 순서를 프로그램 의미에 부합하도록 하는 역할을 한다.

2. MPP의 기본 문제 : 지연시간 및 동기화

MPP는 수많은 노드들로 구성되어 있으므로 이들 사이의 통신에서는 필연적으로 지연시간 문제가 야기된다. 또한 단일 작업을 병렬로 처리하는 경우, 각 병렬 실행 단위들의 동기화가 필수적이다.

2.1 지연시간

지연시간은 프로세서에 의해 요청(request)이 발생된 후 해당 응답(response)이 올 때까지의 경과된 시간을 말한다. 가령, 프로세서에 의해 load, store와 같은 메모리 참조 명령이 시작되면, 읽기 또는 쓰기 요청이 메모리로 전달된다. 메모리 지연시간은 이러한 요청이 발생된 후 메모리로부터 완료 응답이 올 때까지의 시간에 해당된다. 단일프로세서 시스템에서 메모리 지연시간은 DMA에 의한 작업이 없는 한 고정적이다. 그러나 다중프로세서 시스템에서는 여러개의 프로세서에서 나온 요청들 간의 충돌로 인해 메모리 지연시간을 예측할 수 없으며, 더우기 프로세서의 수가 많아질수록 일반적으로 지연시간은 증가하게 된다.

버스 기반 다중프로세서 시스템에서 지연시간 증가의 주요한 원인은 버스의 배타적 사용에 있다. 한 프로세서가 메모리 참조를 하기 위해 버스를 사용하고 있는 동안 다른 프로세서는 버스를 사용할 수 없다. 이러한 문제는 버스를 보다 효율적으로 사용하기 위해 파이프라인 기술을 사용하거나 또는 split-phase 방식을 사용함으로써 완화될 수 있다. split-phase 방식은 메모리 참조 요청과 응답 전 기간 동안 버스를 사용하던 기존의 방법과 달리, 별도로 버스 획득을 함으로써 다른 프로세서에게도 버스 사용의 기회를 준다. 각 프로세서마다 캐쉬를 두는 방법은 버스 사용 빈도를 줄임으로써 버스의 포화(saturation)를 막을 뿐 아니라, 캐쉬 내에 원하는 데이타가 있을 경우 지연시간도 줄게 된다.

버스를 기반으로 하는 시스템은 일반적으로 프로세서의 개수가 많지 않은 작은 시스템이다. 수백~수천개의 프로세서를 사용하는 MPP에서는 버스가 아닌 다른 상호연결망을 사용하게 된다. MPP는 다수의 노드들이 상호연결망을 통해 연결되어 있는 형태를 갖는다. 분산 메모리 형태의 MPP에서 각 노드는 [프로세서-메모리]로 이루어지는데, 이 프로세서와 지역 메모리(local memory) 간에는 버스를 사용한다. 하나의 노드는 단 하나의 프로세서를 가질 수도 있지만, 여러개의 프로세서가 하나의 지역 메모리를 공유하는, 마치 버스 기반 다중프로세서 시스템과 같은 구조를 가질 수 있다.

이러한 MPP에서의 지연시간은 지역 메모리 참조의 경우와 원격 메모리(remote memory) 참조의 경우에 현격한 차이가 있다. 지역 메모리 참조는 앞서 언급한 버스 기반 다중프로세서 시스템처럼 예측 불가능성을 띠고 있긴 하지만, 지연시간은 비교적 작은 값에서 한정된다. 그러나 원격 메모리 참조는 이보다 훨씬 큰, 수백~수천 사이클 이상의 지연시간을 갖는다.

MPP에서의 지연시간 문제는 캐쉬를 사용하여 완화시킬 수 있다. 이는 버스 기반 다중프로세서 시스템에서 캐쉬를 사용하는 경우와 유사한 원리다. 그런데, 버스 기반 다중프로세서 시스템에서는 버스를 통한 스누피(snoopy) 캐쉬 일관성 프로토콜을 사용할 수 있었지만, MPP에서는 상호연결망이 버스와 같이 전역적인 방송 특성을 갖고 있지 못하기 때문에 이의 사용만으로는 캐쉬 일관성 문제의 해결이 불가능하다. 원격 노드에 있는 복사본에 대해 일관성을 유지하기 위해 디렉토리 기반 프로토콜들[2][3][1]이 사용되는데, 여기에 대해서는 3절에서 다루기로 한다.

원격 메모리 참조에 캐쉬를 사용하려면 시스템 전체가 단일 전역 주소 공간을 가져야 한다. 그렇지 않고 주소 공간이 각 노드별로 분할되어 있는 경우, 메시지 전달을 통해 노드간의 명시적 통신이 이루어진다. 메시지 전달 방식은 원격 메모리에 대한 복사본을 사용하지 않기 때문에 캐쉬 일관성 유지가 필요없다. 그러나, 캐쉬를 통해 구현된 공유 메모리 방식과는 달리 프로그래머가 단일 주소 공간을 갖지 못하기 때문에, 프로세스 사이에 복잡한 자료 구조를 넘겨 주기 쉽지 않고 코딩에도 어려움이 따른다. 공유 메모리 방식과 메시지 전달 방식은 모두 피할 수 없는 지연시간 문제를 갖는데, 특히 메시지 전달 방식은 캐쉬를 사용한 지연시간 단축을 기대할 수 없으므로 메시지의 생성 및 전달 방식에 주의를 기울여야 한다.

지연시간 문제는 캐쉬를 사용한 단축 효과 이외에 다중스레드[8]를 이용한 감춤 효과를 통해 해결할 수 있다. 다중스레드 구조에서는 하나의 스레드가 수행중에 긴 지연시간이 유발될 경우(캐쉬 미스가 나거나 또는 원격 노드에 대한 메시지 전달), 다른 스레드로 문맥 전환함으로써 프로세서의 이용률을 높인다. 이에 대해서는 뒤에서 보다 자세히 다루기로 한다.

지연시간은 노드간의 데이타 전송 시간(transport time)에 상당 부분 기인한다. 그러나 한편으로는 많은 상업용 병렬·분산 시스템에서 노드간 통신 오버헤드가 지연시간의 많은 부분을 차지한다. 즉, 프로세서는 네트워크 프로토콜과 네트워크 인터페이스를 운용하는데 노드간 실제 전송보다 더 많은 사이클을 소비한다.

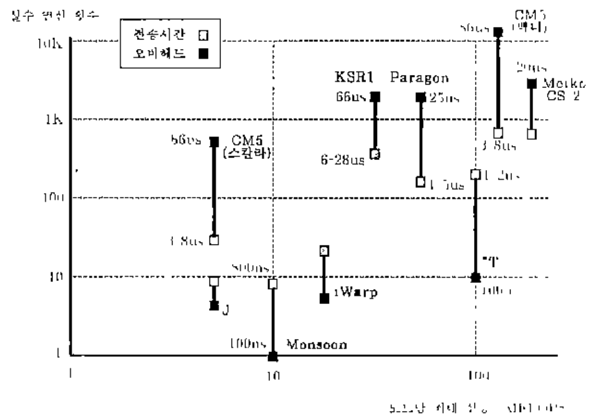


그림 1 전송 시간과 오버헤드 비교

그림 1은 몇몇 상업용 컴퓨터와 연구용 컴퓨터에서 단일 메시지의 네트워크 처리 오버헤드와 전송 시간과의 상대적 비교를 보여준다[9]. 오버헤드는 송신지 프로세서가 패킷을 형성해서 상호연결망으로 보낼 때까지 걸린 시간과 수신지에서 패킷을 받아서 이를 다루는 사용자

코드가 실행될 때까지의 시간을 합한 것이다(사용자 코드의 실행 시간은 제외). 전송 시간은 패킷의 헤더가 송신지를 떠났을 때부터 패킷의 마지막 부분이 수신지에 도착할 때까지의 시간을 말한다. 이 그림은 여러 컴퓨터들을 노드당 최대 실수 연산 성능(FLOPS)별로 분류하여 오버헤드와 전송시간과의 관계를 보여주고 있다.

전송 시간을 실시간(sec)으로 나타내기 보다는 해당 컴퓨터에서 소요되는 실수 연산 회수로 표시하였는데, 이는 지연시간이 차지하는 비중을 보다 분명히 알 수 있게 해준다. 이 척도는 프로세서가 통신에 보내는 시간이 얼마나 많은 유용한 계산을 할 수 있는 시간인가를 나타내고 있다. 상업용 컴퓨터에서 통신 오버헤드는 전적으로 전송 시간을 능가한다[10]. 이 오버헤드의 대부분은 커널 모드 네트워크(Paragon [10], Meiko)를 위한 사용자와 커널간의 상호작용, 최적화되지 않은 메시지 라이브러리 소프트웨어(CM-5[CM5]), 내장(built-in) 통신 메카니즘과 메시지 전달의 부조화(KSR-1[1]) 등에 기인한다. active message[11] 개념은 사용자 모드 네트워크 인터페이스를 제공하는 컴퓨터들에 대해 소프트웨어 오버헤드를 상당히 줄여줄 수 있다. 하지만 소단위(fine-grain) 메시지 전달에 대해서는 여전히 상당한 오버헤드를 차지한다.

대조적으로 연구용 컴퓨터들은 훨씬 나은 통신 성능 수준을 보여준다. 이는 집적된 네트워크 인터페이스가 프로세서 구조에 밀결합되었기 때문이다. 그렇지만 이들 프로세서는 특성화되어 있어 호환성 문제가 따르고 또한 개발 인력, 비용 등에 있어 상업용 프로세서를 따라가기 힘들다. 실제로 이들이 상업용 마이크로프로세서의 성능을 능가하기는 어렵기 때문에 상업용 수퍼스칼라 마이크로프로세서로 병렬 컴퓨터를 구성하는 것이 추세이다.

2.2 동기화

병렬 실행을 위해서 프로그램은 태스크(task)라고 불리는 여러개의 기본적인 단위들로 분할되어야 한다. 일반적인 병렬 프로그래밍 모델에서는 이들 태스크를 동적으로 생성하고 태스크가 데이타를 생성 또는 소비함으로써 특정 작업을 마친 후에는 또한 동적으로 소멸하는 것을 전제한다. 동기화는 이들 태스크간의 시간적 순서를 유지하는 것을 의미한다.

다중프로세서 시스템은 다양한 동기화 자원들을 다룰 수 있는 메카니즘들을 제공하는데, 대개 간단한 프리미티브들이 하드웨어에 의해 직접적으로 제공된다. 이들 프리미티브들은 마이크로코드나 소프트웨어에서 구현되는 복잡한 메카니즘의 상호배제(mutual exclusion)를 실현하는데 사용되는 기초적인 도구이다.

최소의 하드웨어 지원은 아마도 Test&Set과 같이 원자 동작(atomic operation)이 이루어지는 명령어를 제공하는 것일 것이다. 소프트웨어는 잠금(lock)이 풀릴 때까지(잠금 변수＝0) Test&Set을 반복한다. 이것은 프로세서로 하여금 잠금 변수를 검사하는 루프를 계속 돌게 함으로써 프로세서 사이클의 낭비와 메모리 참조량의 증가를 유발한다. 이런 busy-waiting 방식에 의존하는 형태를 spin-lock이라 부른다. 반면, suspend-lock 또는 sleep-lock[sync, ordering 1번]에서는 잠금 변수의 계속적인 검사 대신 인터럽트에 의존함으로써 메모리 참조를 줄인다. 여기서는 잠금을 얻는데 실패하면, 더이상 잠금 변수를 검사하기 위해 메모리 참조를 하지 않고 잠금을 푸는 프로세스가 인터럽트를 통해 잠금을 돌려줄 때까지 기다린다. 이 방법은 메모리 참조를 줄일 수는 있지만, 프로세서 사이클은 여전히 낭비된다.

Test&Set과 유사한 프리미티브로 IBM 370 구조에서 사용된 Compare&Swap이 있다. Test&Set이나 Compare&Swap 모두 read-modify-write의 원자 동작에 기초하고 있는데, 이들은 모두 공통적인 성능 문제를 안고 있다. 만일 n개의 프로세스가 동시에 critical section에 진입하려고 시도한다면, 그중에 기껏해야 하나만 성공하게 될 것임에도 불구하고 메모리 시스템은 n개의 잠금 동작을 경험하게 될 것이다. 이런 문제점은 NYU Ultracomputer[12]와 RP3에서 Fetch&Add를 통해 해결이 모색되었다. Ultracomputer에서 Fetch&Add를 사용한 *n-way* 동기화의 복잡성은 n 값에 무관하다. 이 프리미티브의 실행은 프로세서와 메모리 사이

에 있는 결합네트워크(combining network)라 불리는 상호연결망에 의해 분산적으로 이루어진다. n개의 프로세서가 동시에 같은 메모리 워드에 대해 Fetch&Add를 실행하면, 메모리의 갱신은 단 한번만 이루어지고 각 프로세서에는 더해진 유일한 값들이 상호연결망에 의해 제공된다. 돌아온 값들은 n개의 요청에 대해 순차적으로 부여된 순서와 같다. 프로세서와 메모리의 입장에서 보면 그 결과는 n개의 Fetch&Add를 순차 실행한 것과 일치하지만, 메모리에서의 수행은 단 한번만 이루어진다. 예를들어 두개의 프로세서 i, j에 의해 각각 Fetch&Add(A, V_i)와 Fetch&Add(A, V_j)가 실행되면, 프로세서 i, j에는 각각 [A], [A]$+V_i$ 또는 [A]$+V_j$, [A]가 결과로 돌아온다. 결과적으로 Fetch&Add는 여러 프로세스의 큐에 대한 접근 또는 코드는 같지만 서로 다른 데이타 세그먼트를 갖는 프로세스들로의 fork등에 있어 매우 효과적이다. 결합 네트워크는 많은 프로세서가 한 메모리로 요구를 하는 경우 메모리가 경쟁 지점이 되는 것을 막을 수 있지만, 네트워크 요소에 계산 능력을 요구하므로 비용 문제가 따른다.

HEP[13]에서는 메모리 워드가 empty 또는 full의 상태를 갖는다. empty인 메모리에 읽기 요구가 있으면 이 요구는 메모리에 쓰기가 있을 때까지 보류된다. 차후에 쓰기가 일어나면 읽기 요구에 응답한다. full인 메모리에 대한 요구는 바로 응답하고 태그를 empty로 한다. 마찬가지로 full인 메모리에 대한 쓰기는 empty가 될 때까지 보류되었다가 실행된다. 이러한 메카니즘은 생산자-소비자 관계처럼 하나의 프로세스는 다른 프로세스가 메모리에 쓰기를 할 때까지 기다리므로 프로세스간의 동기화를 위해 사용될 수 있다. HEP에서 하나의 프로세서에는 여러개의 문맥이 동시에 유지되는데, 동기화가 실패하면 곧바로 다른 문맥으로 전환되므로 동기화에 낭비되는 프로세서 사이클은 적다. 그러나 태그의 관리는 여전히 프로그래머 또는 컴파일러에 대한 부담으로 남는다.

I-structure[14] 메모리는 생산자-소비자를 위한 효과적인 동기화를 제공한다. I-structure의 각 워드는 empty, present, deferred 중의 하나의 상태를 갖는다. present 또는 empry는 값이 차 있거나 또는 비어 있는 상태를 나타내며, HEP에서처럼 present 워드만을 읽을 수 있고 empty 워드에만 기록할 수 있다. deferred는 empty 워드에 대해 최소한 한번 이상의 읽기 동작이 시도됐을 때의 상태이다. 이 상태에서 나중에 값이 쓰여지면 "deferred"된 모든 읽기 요청들이 만족된다. "deferred"된 읽기 요청을 한 요청자에 대한 정보는 특별한 장소에 "deferred list"로 유지된다. I-structure는 함수형 언어(functional language)의 실행에 있어서 non-strictness를 제공할 수 있고 또한 non-busy-waiting에 사용될 수 있는 장점을 갖는다.

동기화 비용은 동기화 명령어들을 수행하는데 걸리는 시간 뿐 아니라 문맥 전환(*context* switching) 시간까지도 포함된다. 동기화 사건을 기다리는 태스크는 조건들이 바로 만족될 수 없는 경우에 다른 태스크로 문맥 전환함으로써 프로세서 사이클의 낭비를 줄일 수 있다. 마치 I/O 동작과 같은 긴 수행시간을 기다리지 않고 다른 프로세스로 문맥 전환을 함으로써 프로세서의 이용률을 높이는 것과 같다. 이 경우 문맥 전환은 동기화가 없는 순차 실행에서는 발생하지 않는 것으로써 동기화의 효율을 위해 사용된 것이므로 동기화 비용에 해당된다. 문맥 전환 비용은 비교적 큰 비중을 차지할 수 있는데, 이는 태스크와 연관된 상태(가령, 레지스터의 내용)들을 저장하는 시간이 포함되기 때문이다. I/O 동작의 경우처럼 문맥 전환의 빈도가 비교적 적은 경우에는 별로 부담이 되지 않지만, 병렬 실행 환경에서 병렬성이 커지면, 동기화가 빈번하게 일어나고 또한 이에 따르는 문맥 전환 비용도 적지 않게 된다. 따라서 보다 많은 병렬성의 활용과 동기화 비용 문제는 서로 상충하게 된다.

동기화 기법은 시스템의 구조에 따라 적절히 사용한다. Test&Set, Compare&Swap, 세마포어(semaphore)와 Fet&Add는 공유 메모리를 사용하는 시스템에서 사용되는 동기화 기법이고 메세지 전달은 그렇지 않은 시스템에서 사용된다. 동기화로 인하여 전체 시스템의 성능이 제약을 받는 경우는 시스템 특성에 따라 추가적인 하드웨어를 필요로 할 수 있다. 같은 공

표 1 다중프로세서와 다중컴퓨터 비교

항목	다중프로세서	다중컴퓨터
주소 공간	단일 전역 주소 공간	노드별 주소 공간
운영체재	일반적으로 단일 운영체제	노드별로 운영체제 커널 보유
작업 큐	단일 공통 작업 큐	작업이 노드들에 분산
작업 크기	임의 크기(시스템 메모리 크기)	노드 메모리 크기에 제한됨
데이타 전달 방식	공유 메모리에 직접 접근	명시적 메시지 전달
데이타 접근의 대칭성	대칭적	지역, 원격 메모리별로 접근 방식이 달라짐
원격 데이타에 대한 접근	자동적	주소 변환, 메시지 전달 필요

유 메모리 접근의 문제를 해결한 NYU Ultracomputer의 결합 네트워크가 좋은 예다.

시스템의 병렬 수행 단위에 따라 다른 동기화 기법들이 사용될 수 있는데, 병렬 수행 단위가 작은 경우는 하드웨어 기법을 사용하고 함수나 프로그램과 같이 큰 경우에는 소프트웨어 기법을 사용한다.

3. 공유 메모리 구현 기법

MIMD에서 흔히 고려되는 두가지 프로그래밍 모델로 공유 메모리(shared-memory) 방식과 메세지 전달 방식(message-passing)이 있다. Gordon Bell의 분류에 의하면 공유 메모리 방식은 다중프로세서에서, 메세지 전달 방식은 다중컴퓨터에서 보통 사용된다[22]. 이 둘 사이의 중요한 차이는 단일 전역 주소 공간(single global address space)을 갖느냐의 여부이다. 다중프로세서 시스템이 단일 주소 공간을 갖는데 비해 다중컴퓨터 시스템은 그렇지 않다. 다중프로세서와 다중컴퓨터를 비교하면 표 1과 같다. 현재는 이 둘 사이의 구분이 모호해지는 경향이 있다. 가령, FLASH[21]에서는 전역 캐쉬를 사용한 공유 메모리와 메시지 전달, 두가지 모두를 지원하고 있다.

앞의 표에서 보듯이 일반적으로 메세지 전달 방식이 공유 메모리 방식에 비해 자료 접근을 처리하기 어렵다. 프로그래머에게는 전체 메모리가 하나의 주소 공간으로 되어 있는 공유 메모리 방식이 보다 이해하기 쉬울 것이다. 따라서 물리적으로 하나의 주소 공간을 갖는지의 여부에 관계없이 어떤 형태로든 프로그래머에게 가상적인 전역 공유 메모리 형태를 보여주는 것이 현재의 추세이다.

일반적으로 중앙 메모리(centralized memory) 형태는 상호연결망에서의 병목 현상으로 확장성을 갖기 어려우므로 분산 메모리(distrubuted memory) 형태를 가질 필요가 있다. 분산 메모리 형태에서 공유 메모리를 지원하는 방법으로는 전역 캐쉬(global cache)를 사용하는 방법(다중프로세서)과 메시지 전달을 이용하는 방법(다중컴퓨터)이 있다. 전자가 하드웨어적인 캐쉬 메카니즘에 의존하고 있는데 반해, 후자는 소프트웨어적으로 원격 노드에 대한 접근을 처리하고 있다. 어느 것이나 프로그래머에게는 하나의 공유 메모리 공간이 제시된다. 이제 이 두가지 방법에 대해 좀더 자세히 살펴보기로 한다.

3.1 메시지 전달을 통한 공유 메모리 구현

메시지 전달은 프로그래머에게 공유 메모리를 제공하는 것을 가능하게 해준다. 가령, 프로세서 i가 프로세서 j의 메모리에 있는 데이타를 필요로 할 경우, 프로세서 i는 이를 요청하는 메시지를 프로세서 j에게 보낸다. 요청 메시지를 받은 프로세서 j는 요청된 데이타를 프로세서 i에게 응답 메시지를 통해 보낸다.

요청을 보낸 프로세서 i는 프로세서 j가 응답을 보내올 때까지 아무것도 하지 않고 단지 기다릴 수 있다. 그러나 이 경우 프로세서 j 역시 프로세서 i의 응답을 기다리고 있는 상태일 수 있기 때문에 deadlock 가능성이 있다. 따라서 원격 노드에서 온 메모리 접근 요청을 담당하

는 별도의 장치를 두거나 또는 프로세서가 주기적으로 다른 노드에서 온 요청들을 처리해주어야 한다. 전자는 *T[15], Cray T3D[16]에서 사용되고 있다. 이 방법은 데이타가 이용 가능하기만 하면 항상 올바르게 작동한다. 주기적인 폴링(polling)을 하는 후자에는 코드 발생의 어려움과 성능 저하 문제 등이 있다.

요청한 데이타가 아직 생성되지 않은 경우(가령, 앞에서 살펴본 HEP이나 I-structure에서 메모리 워드가 empty인 경우), 현재 읽기 요청을 한 문맥을 중단하고 다른 문맥으로 전환하는 것이 필요하다. 그렇지 않을 경우 데이타를 생성할 문맥의 실행을 계속 막을 수 있기 때문에 deadlock이 발생할 수 있다. 요청과 응답 메시지는 연속(continuation)이라 불리는 정보를 담아야 하는데, 이는 요청한 데이타가 돌아왔을 때 이를 수신하여 처리할 문맥을 나타낸다.

이와 같이 요청과 응답이 분리되어 스케줄되는 split-phase 방식은 몇가지 좋은 특성들을 갖는다. 우선, 계산과 통신이 병행하여 이루어지기 때문에 지연시간 감춤 효과를 얻을 수 있다는 점을 들 수 있다. 일단 요청을 보낸 프로세서는 응답이 올 때까지 기다리지 않고 문맥 전환을 하여 다른 유용한 일을 할 수 있다. 사실 MPP에서 다른 노드의 데이타를 읽는데는 요청과 응답을 합하여 두번의 메시지 송신, 두번의 상호연결망 통과, 두번의 메시지 처리기(handler) 실행이 포함되어 수백~수천 사이클이 걸린다. split-phase 방식은 이 지연시간을 활용할 수 있지만, 문맥 전환이라는 부담이 따른다. 만일 문맥 전환 시간이 원격 노드 접근 시간보다 더 크다면, 이의 효용성은 없다. 결국 메시지 전달을 통해 공유 메모리를 구현하는 방식에서는 빠른 문맥 전환과 프로세서/네트워크 인터페이스의 성능이 중요한 역할을 한다.

앞선 언급한 바와 같이 다른 노드로부터의 메모리 접근 요청을 처리해주는 별도의 하드웨어를 갖고 있지 않은 경우에는 문맥 전환이 추가적으로 일어나므로 부담이 더 커진다. 원격 노드는 현재 수행중이던 문맥에서 메모리 접근 요청을 처리하는 문맥으로 전환해야 하기 때문이다. 이러한 이유로 특별한 기능을 갖지 않는 일반적인 상업용 마이크로프로세서만을 사용하여 메시지 전달에 기반을 둔 공유 메모리를 구현하는 것은 비효율적일 가능성이 높다. 따라서 문맥 전환의 빈도를 줄이고 워드 단위의 메모리 접근보다는 가급적 긴 메시지 패킷을 사용하도록 컴파일러가 코드를 발생시키는 것이 성능에 중요하다.

3.2 전역 캐쉬를 사용한 공유 메모리 구현

전역 캐쉬가 지원되는 경우 원격 노드에 있는 데이타일지라도 프로세서는 마치 지역 메모리에 있는 데이타에 접근하듯이 동작한다. 읽기를 원하는 원격 노드의 데이타가 캐쉬에서 발견되면, 바로 요청이 만족되고 실행이 계속된다. 그러나 캐쉬에 없는 경우에는 일단 지역 메모리 버스로 요청이 전파되고, 메모리 버스 인터페이스에 있는 특정 장치가 전역 주소를 인식하여 자동적으로 원격 노드와 통신, 원하는 값을 가져온다. 이러한 동작은 프로세서에게 투명하기 때문에, 프로세서는 필요로 하는 데이타가 지역 메모리에 있는지 또는 원격 메모리에 있는지에 대해 간여할 필요가 없다.

필요로 하는 원격 데이타가 캐쉬에 있으면 빠른 시간에 프로세서의 메모리 요청을 만족시킬 수 있지만, 그렇지 않을 경우 상황은 메시지 전달 방식과 같아진다. 이 경우 메모리 접근 시간은 원격 노드로부터 데이타를 가져오는데 걸리는 긴 지연시간을 포함하게 된다.

한편, 최근의 마이크로프로세서는 2개 이상의 명령어를 동시에 이슈(issue)하는 수퍼스칼라 방식을 채택하는 경우가 많다. 또한 이들 마이크로프로세서는 명령어를 out-of-order로 이슈함으로써 효율을 높이기도 한다. load 명령어가 완료되기 전이라도, 이후에 나오는 명령어들이 이 load에 의해 읽혀질 데이타를 실제로 필요로 하는 명령어가 아니면 계속해서 진행이 가능하다. 또한 앞으로의 마이크로프로세서는 여러개의 메모리 동작들을 동시에 실행할 수 있게 될 것이다. 이러한 특성들은 모두 캐쉬 미스로 인한 원격 노드로의 통신과 병행하여 다른 유용한 작업들을 프로세서가 계속해서 할 수 있음을 의미한다. 이와같은 통신과 계산 작업의 병행성 성취 여부는 프로그램 특성에 크

게 좌우되긴 하지만, split-phase 방식보다는 크게 뒤떨어질 것으로 보인다. 왜냐하면 명령어 수준에서 추구할 수 있는 병렬성은 그다지 크지 않기 때문이다.

캐쉬 미스가 발생했을 때 단순히 기다리기보다는 메시지 전달 방식에서처럼 문맥 전환을 함으로써 긴 지연시간을 활용할 수 있을 것이다. 그러나 이 경우 문맥 전환은 몇가지 문제점을 가져온다. 먼저, 전환된 문맥이 수행되다가 다시 원래의 문맥(캐쉬 미스가 발생했던 문맥)으로 돌아왔을 때, 원격 메모리로부터 가져온 데이타가 캐쉬에 남아 있으리라는 보장이 없다. 왜냐하면 전환된 문맥도 캐쉬를 사용하면서 해당 블럭을 다른 블럭으로 대치할 가능성이 있기 때문이다. 이를 위해서는 캐쉬 블럭에 대한 잠금(lock)이라든지 또는 가져온 블럭을 어떤 다른 버퍼에 보관하는 메카니즘이 있다. 이 문제점을 해결하여 캐쉬 미스시의 문맥 전환을 도입한 컴퓨터로 Alewife[17]가 있다. 또다른 문제점은 다중의 하드웨어 문맥을 제공하지 않으면, 문맥 전환 비용이 매우 크다는 점이다. 왜냐하면 이러한 구조에서 사용되는 코드는 대개 레지스터에 많은 상태를 저장하고 있기 때문에 문맥 전환시에 이들을 모두 대피시켜야 하는 부담을 안고 있다.

가장 큰 문제는 캐쉬 일관성을 유지하는 것이다. 두개 이상의 복사본이 서로 다른 캐쉬에 존재하는 경우, 어느 하나가 갱신을 하면 이 사실을 자동적으로 다른 캐쉬에 인지시킬 수 있어야 한다. 분산 메모리에서 캐쉬 일관성을 유지하는 것은 상당히 복잡한 메카니즘을 필요로 한다. DASH[3], KSR[1], SCI[2], Alewife [17] 등의 컴퓨터들이 이러한 캐쉬 일관성 프로토콜들을 제안하고 있다. 이에 대해서는 다음 절에서 좀 더 고찰해보기로 한다.

3.3 캐쉬 일관성 유지 프로토콜

캐쉬를 사용하는 공유 메모리 다중프로세서 시스템에서는 캐쉬 일관성(coherence) 유지가 필수적이다. 여러 캐쉬가 동일한 주소의 메모리 내용을 동시에 갖고 있으면 메모리 내용이 바뀔 때마다 모든 복사본이 일치하도록 보장하는 방법이 존재해야 한다. 어떤 시스템에서는 공유되는 블럭을 캐쉬할 수 없다고 표시하거나, 프로세서간 프로세스 이동을 제한 또는 금지하는 소프트웨어 기법을 사용한다. 다른 방법은 모든 블럭이 각 프로세서에게 캐쉉될 수 있도록 허용하고 일관성 유지를 위한 프로토콜을 사용한다. 이러한 프로토콜에는 일반적인 상호연결망을 위한 디렉토리 기반 프로토콜(directory-based protocol)과 공유 버스 시스템을 기반으로 하는 스누핑 프로토콜(snooping protocol)이 있다.

스누핑 프로토콜에서는 시스템 내 다른 프로세서들의 버스 트랜잭션을 캐쉬 제어기(cache controller)들이 관찰하고 있다가 자신의 캐쉬 내용과 관련된 트랜잭션인 경우 일관성 유지를 위해 적절한 조치들을 취한다. 시스템에 있는 각 블럭의 상태는 캐쉬 제어기로 분산된다. 스누핑 프로토콜의 대표적인 예로는 Write Once, Synapse N+1, Berkeley, Illinois, Firefly, Dragon 등을 들 수 있다[18].

스누핑 캐쉬 프로토콜들은 공유 버스에 기반을 두고 있기 때문에 확장성에 문제가 있고, 분산 메모리 형태를 지원할 수 없다. 확장성있는 전역 캐쉬를 지원하고자 제안된 디렉토리 기반 프로토콜로서 대표적인 것으로는 SCI(Scalable Coherent Interface)와 DASH에서 사용한 프로토콜 등이 있다.

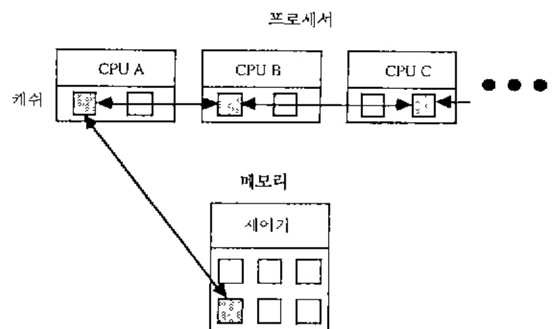


그림 2 SCI의 분산된 캐쉬 태그

• SCI

SCI는 IEEE 표준 프로젝트에서 정의된 프로토콜로 이중 링크 리스트로 구성된다. 메모리와 캐쉬의 각 블럭에는 태그가 있고 메모리의 태그는 메모리 블럭을 공유하는 캐쉬 리스트의

첫번째 캐쉬를 가리킨다. 캐쉬의 태그는 같은 블럭을 공유하는 앞뒤의 캐쉬를 연결하기 위해 사용된다.

캐쉬 A가 메모리 읽기 요구를 하면 메모리 제어기는 태그를 살핀다. 이미 읽은 다른 캐쉬가 없으면 데이타를 캐쉬 A에게 주고 태그를 A로 정한다. 이미 태그에 캐쉬 B를 가진 경우는 캐쉬 B의 포인터를 캐쉬 A에게 준다. 캐쉬 A는 캐쉬 B의 포인터로 캐쉬 B에게 읽기 요구를 하면, 캐쉬 B는 포인터를 캐쉬 A로 수정하고 데이타를 캐쉬 A에 전송한다.

리스트의 헤더만이 배타적인 접근을 위해 다른 리스트를 제거할 수 있으며 캐쉬 A가 다음 캐쉬인 B에 제거 명령을 내리면 캐쉬 B는 그 블럭을 무효화하고 자신의 다음 포인터를 캐쉬 A에게 넘긴다. 그러면 다시 캐쉬 A는 이 포인터로 다음 캐쉬에서 그 블럭을 제거한다. 모두 제거된 후에야 쓰기를 할 수 있다. 이 프로토콜에서 리스트의 길이가 긴 경우가 드물다고는 하지만 각 요구가 상호연결망을 경유하게 되므로 오랜 시간이 소요된다.

• DASH

DASH 프로토콜에서는 클러스터 내에서 스누핑 캐쉬 프로토콜을 사용하고 클러스터간에는 디렉토리 기반 프로토콜을 사용한다. DASH 프로토콜에서 메모리 블럭은 uncached-remote (다른 원거리 클러스터가 복사본을 가지지 않는 경우), shared-remote (다른 원거리 클러스터들이 수정되지 않은 복사본을 가지는 경우)와 dirty-remote (한 원거리 클러스터가 수정된 복사본을 가지는 경우) 등 세 가지 상태를 갖는다.

프로세서가 생성한 읽기 요구가 클러스터 내에서 만족되지 않을 경우 읽기 요구는 그 블럭을 가지는 홈 클러스터(home cluster)로 보내진다. 홈 클러스터는 그 블럭의 상태를 살펴서 uncached-remote나 shared-remote인 경우는 홈 클러스터가 직접 요구한 클러스터에 블럭을 전송한다. 그리고 요구한 클러스터가 메모리 블럭을 가지고 있음을 디렉토리에 기록한다. 그렇지 않고 dirty-remote인 경우에는 그 블럭을 보유하고 있는 소유 클러스터(owing cluster)로 읽기 요구를 전송한다. 소유 클러스터는 읽기 요구에 따라 그 블럭을 요구 클러스터에게 바로 전송한다. 동시에 홈 클러스터에 수정된 값을 기록한다. 소유 클러스터는 블럭을 dirty에서 read-only로 표시하고 홈 클러스터는 블럭을 shared-remote 상태로 전이시키고 디렉토리를 갱신한다. 앞의 SCI와 비교하면 읽기

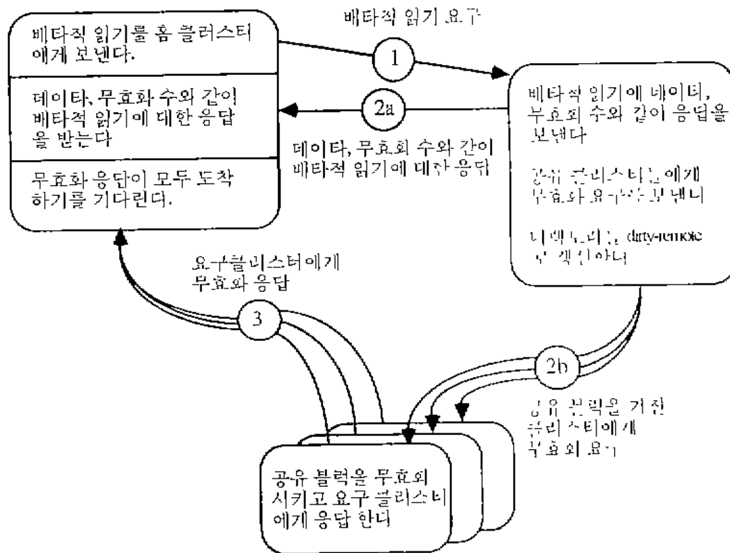


그림 3 원거리 메모리에서 shared-remote상태인 경우

요구에서 DASH가 더 나음을 알 수 있다. SCI에서는 읽기 요구가 4번의 전송으로 완료되나 DASH에서는 3번의 전송으로 완료된다.

블럭에 쓰기를 하려면 캐쉬는 배타적인 권한을 가져야 한다. 블럭이 클러스터 내 메모리라면 캐쉬 스누핑 프로토콜로 하면 되지만 원거리 클러스터에 있는 메모리라면 배타적 읽기(read-exclusive) 요구를 해야 한다. 배타적 읽기 요구도 읽기 요구와 비슷하다. 먼저 홈 클러스터로 요구가 전해지고 그 블럭의 상태가 uncached-remote나 shared-remote인 경우는 홈 클러스터가 직접 요구한 클러스터에 블럭을 전송하고, 상태를 shared-remote로 바꾼다. shared-remote인 경우는 그 블럭을 가지는 각 클러스터에 무효화하도록 지시하고 요구 클러스터는 데이타와 무효화 응답(ACK)의 수를 받는다(그림 3). 주어진 수만큼의 응답을 받으면 쓰기가 가능해진다. 다른 원거리 클러스터들이 무효화 요구를 받으면 응답을 요구 클러스터에게 보낸다. dirty-remote인 경우는 소유 클러스터로 배타적 읽기 요구를 전송하고 소유 클러스터는 그 블럭을 무효화시키고 데이타와 소유권을 요구 클러스터에게 보낸다. 동시에 홈 클러스터에게 소유권 갱신 요구를 보낸다. 이 요구를 받은 홈 클러스터는 요구 클러스터에게 응답을 보내 소유권이 바뀌었음을 알린다. SCI와 비교하면 블럭을 공유한 경우, DASH가 훨씬 좋은 성능을 보일 것으로 예상된다. SCI는 리스트를 따라 각 노드에 무효화를 차례로 보내지만 DASH는 한꺼번에 보내고 그 응답만 기다리면 된다. 공유 리스트가 길지 않은 경우는 그다지 차이가 없다. 블럭의 공유가 많지 않은 응용 분야인 경우에는 SCI도 유용한 프로토콜이 된다.

4. 상호연결망

MPP의 상호연결망은 많은 노드들이 서로 협력하여 단일 작업을 효율적으로 수행할 수 있도록 하기 위하여 노드들 간의 고속 통신을 보장하여야 한다. 이러한 상호연결망은 위상, 동작 모드, 스위칭 방법, 제어 방법 등에 따라 여러 가지가 존재한다.

현재 제안되었거나 구현된 상호연결망에는 일렬 배열(linear array), 링(ring), 완전 연결형(completely connected), 트리(tree), 팻 트리(fat tree), 성형(star), 메쉬(mesh), 토러스(torus), 시스톨릭 배열(systolic array), 하이퍼큐브(hypercubes), 큐브 연결 사이클(cube connected cycles) 등의 정적 위상을 갖는 상호연결망과 버스, 다단계 네트워크(multistage network), 크로스바 스위치(crossbar switch) 등의 동적 위상을 갖는 상호연결망이 있다.

정적 위상을 갖는 상호연결망은 직접 연결된 경로를 사용하며, 한번 마련된 경로는 변경시킬 수 없다. 이런 형태의 상호연결망은 통신 유형을 예측할 수 있는 경우에 적합하다. 반면에 동적 위상을 갖는 상호연결망에서는 가능한 모든 유형의 통신을 할 수 있기 때문에 보다 범용적인 시스템 구축에 적합하다. 이러한 상호연결망에서는 동적인 연결을 위하여 경로 뿐만 아니라 스위칭 모듈 또는 중재기(arbiter)가 있어야 한다.

위에서 열거된 상호연결망 중 일렬 배열과 버스, 트리는 MPP를 위한 충분한 대역폭을 제공하기 어렵고, 완전 연결형은 비용이 많이 들고, 성형과 시스톨릭 배열은 구현에 어려운 문제가 있다. 여기에서는 MPP에 사용 가능한 구조로써 정적 위상을 갖는 상호연결망인 링, 팻 트리, 메쉬, 토러스, 하이퍼큐브, 큐브 연결 사이클 등과 동적 위상을 갖는 상호연결망인 다단계 네트워크, 크로스바 네트워크 등에 대하여 살펴본다.

• 링(ring)

링 구조는 그림 4와 같으며 모든 노드의 차수는 2로 동일하며, 단방향성과 양방향성 두 가지가 있다. 노드가 N 개 장착된 시스템의 경우 노드간 최대 거리는 단방향성일 경우 N이며, 양방향성일 경우 N/2 이 된다.

IBM의 토큰 링이 이런 구조를 갖는데, 토큰을 갖고 있는 노드에서 발생된 메시지는 링을 따라 회전하여 목적지에 도달한다. CDC Cyberplus 병렬 처리기와 KSR-1에서 파이프라인 또는 패킷 교환 방식을 사용하는 링 구조 상호연결망을 사용하고 있다.

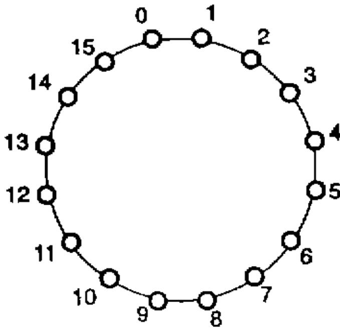


그림 4 링 구조

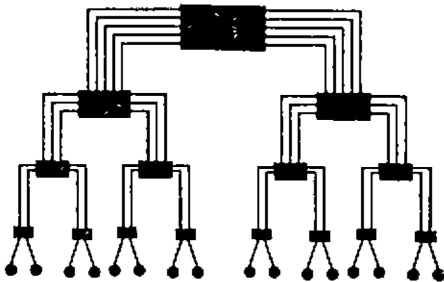


그림 5 팻 트리 구조

• 팻 트리(fat tree)

k-수준(level)을 갖는 완전 균형 이진 트리(completely balanced binary tree)의 노드 수는 N = 2k-1이며, 각 노드의 최대 거리는 2(k-1)이다. 이진 트리는 노드의 차수가 고정되어 있으므로 확장성이 뛰어나다. 팻 트리는 그림 5와 같이 트리 구조의 일종으로 1985년 Leiserson[19]에 의하여 제안되었다. 일반적인 이진 트리의 가장 큰 문제점은 루트 노드에 가까운 경로에서 병목 현상이 발생한다는 것인데, 팻 트리에서는 이러한 문제점의 해결을 위하여 루트 노드에 가까이 가더라도 대역폭의 감소가 현저하지 않도록 설계한 것이다. 현재 4진 팻 트리가 CM-5[6]에서 사용되고 있다.

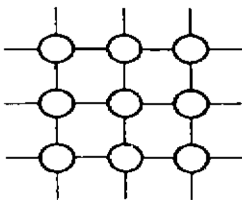
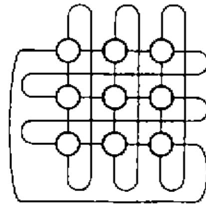


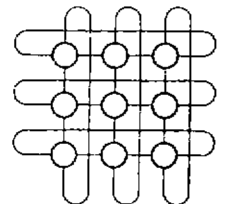
그림 6 2차원 메쉬 구조

• 메쉬(mesh)

k-차원의 메쉬 구조는 N = nk 개의 노드를 갖고 있으며, 각 노드의 차수는 2k이고 노드간 최대 거리는 $k \times (n\text{-}1)$이다. 순수한 메쉬 구조는 대칭적이지 않고, 경계 부분에 있는 노드의 차수가 중간 부분 노드의 차수보다 적다. 2차원 메쉬 구조를 나타내면 그림 6과 같다.



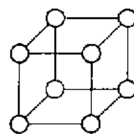
(a) Illiac 메쉬



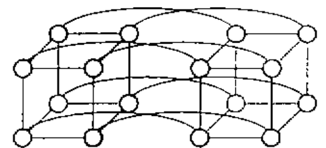
(b) 토러스

그림 7 Illiac 메쉬와 토러스

Illiac IV에서 채택한 메쉬 구조는 2-차원으로 경계 부분 노드들을 서로 연결하여 대칭성을 갖도록 하였다. 일반적으로 $n \times n$ Illiac 메쉬에서 노드간의 최대 거리는 n-1로 순수 메쉬 구조의 절반에 해당한다. 메쉬 구조를 변형하여 노드간 최대 거리를 더욱 감소시킨 구조로 토러스(torus)가 있다. 이 구조는 링과 메쉬를 결합하였으며 대칭적인 형태를 갖는다. 일반적으로 $n \times n$ 이진 토러스에 사용되는 노드의 차수는 4이고 노드간 최대 거리는 2 N/2 이다. 그림 7에서 (a)는 Illiac 메쉬이고, (b)는 토러스를 나타낸 것이다.



(a) 3차원 하이퍼큐브



(b) 4차원 하이퍼큐브

그림 8 하이퍼큐브 구조

메쉬 구조는 Illiac IV, MPP, DAP, Intel Paragon, DASH[4] 등에서 상호연결망으로 채택되었으며 현재 많이 사용되고 있다.

• 하이퍼큐브(hypercubes)

일반적으로 k-큐브에는 N = $2k$개의 노드가 k 차원으로 배열되어 있으며, 각 노드의 차수와 각 노드간 최대 거리는 동일하게 k이다. 차원에 따라 노드의 차수가 증가하기 때문에 하이퍼큐브 구조를 갖는 상호연결망은 확장성이 있다고 보기 어렵다. 그러나 하이퍼큐브 구조는 이진 트리, 메쉬 등의 구조를 내포할 수 있는 융통성을 가지고 있다. 그림 8은 3차원 하이퍼큐브 구조와 4차원 하이퍼큐브 구조를 나타낸 것이다.

1980년대에 이진 하이퍼큐브에 대한 연구 및 개발이 활발히 이루어졌다. 이 시기에 Intel iPSC/1, iPSC/2 그리고 nCUBE에서 상호연결망으로 하이퍼큐브 구조를 채택하였다. 그러나 하이퍼큐브는 확장성에 문제가 있고, 높은 차원으로 구성할 때 패키징의 어려움으로 인해 점차 사용이 감소되는 추세에 있다. 예를 들어, CM-2에서는 하이퍼큐브 구조를 채택하였지만 CM-5에서는 팻 트리 구조가 선택되었고, Intel 사는 iPSC/1, iPSC/2 등에서 하이퍼큐브 구조를 채택하였지만 Paragon에서는 메쉬 구조를 채택하였다.

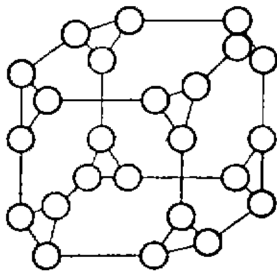


그림 9 큐브 연결 사이클

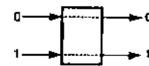
• 큐브 연결 사이클(Cube Connected Cycles)

큐브 연결 사이클은 하이퍼 큐브를 변형시킨 구조로서, 그림 9에 나타난 바와 같이 3-큐브 연결 사이클은 3-큐브를 변형시킨 구조이다.

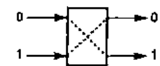
k-큐브를 k-큐브 연결 사이클로 변형시키려면 k-큐브의 꼭지점을 k 개의 노드를 갖는 링으로 대체하면 된다. 이런 방법으로 $2k$ 개의 노드를 갖는 k-큐브는 $k \times 2k$ 개의 노드를 갖는 k-큐브 연결 사이클으로 변형된다. k-큐브에서 두 노드간 최대 거리가 k이지만 k-큐브 연결 사이클에서 두 노드간 최대 거리는 $2k$가 된다. 여기서 k-큐브 연결 사이클이 k-큐브보다 많은 노드를 가지고 있다는 점을 고려한다면 최대 거리의 증가는 그리 큰 것이 아니다. 큐브 연결 사이클의 가장 큰 장점은 큐브 연결 사이클의 차수와는 무관하게 노드의 차수가 3으로 고정되어 있다는 점이다.

• 다단계 네트워크(Multistage Network)

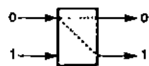
다단계 네트워크는 MIMD 방식과 SIMD 방식 모두에서 사용 가능하며, 우수한 확장성을 가지고 있다. 다단계 네트워크의 통신 지연 시간은 log n의 비율로 증가한다. 일반적으로 그림 10과 같이 스위치 모듈이 여러 단계에 걸쳐서 배열된다. 각 단계들 간에는 고정된 경로로



(a) 통과

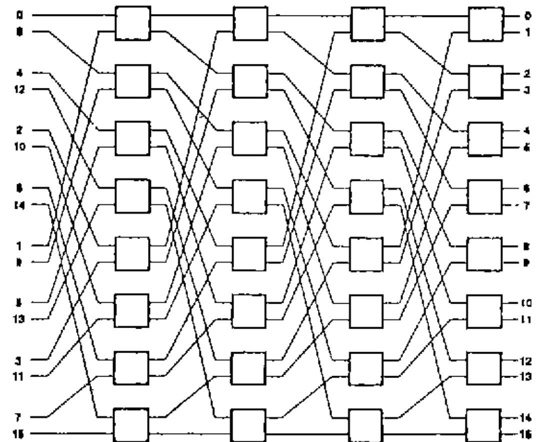


(b) 교차



(c) 위 브로드캐스트

(d) 아래 브로드캐스트



(e) 16x16 오메가 네트워크

그림 10 오메가 네트워크

정적으로 연결되어 있고, 스위치 모듈 내에서 동적으로 경로가 설정된다. 각 단계 간의 경로와 스위치 모듈의 기능에 따라 여러 종류의 다단계 네트워크를 구성할 수 있다. 다단계 네트워크의 한 예로 오메가 네트워크(omega network)의 스위치 모듈의 기능과 각 단계 간의 경로는 그림 10과 같다. 다단계 네트워크는 EM-4[20]와 RP3에서 구현되었다.

• 크로스바 네트워크(Crossbar Network)

크로스바 네트워크는 그림 11과 같은 구조를 가진다. 크로스바 네트워크를 이용하면 가장 큰 대역폭과 연결성을 얻을 수 있지만 하드웨어 복잡도(complexity)가 2n의 비율로 증가하므로 구축 비용이 가장 많이 든다. 크로스바 네트워크에서는 각 스위치가 특정 시점에 어떤 경로를 마련할 것인지를 프로그래밍함으로써 전화 연결과 같이 두 노드를 동적으로 연결하는 것이 가능하다. 크로스바 네트워크는 후지쯔사의 VPP500에서 구현되었다.

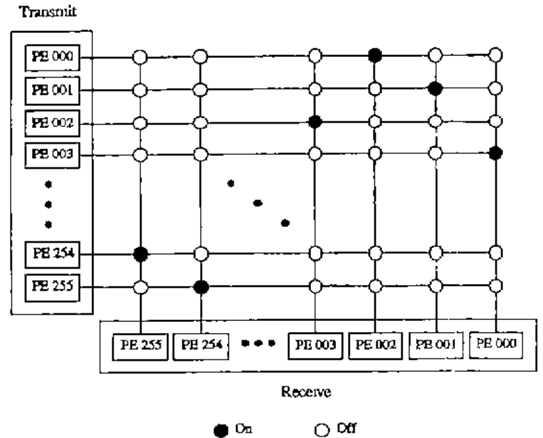


그림 11 크로스바 네트워크

5. 결 론

고성능 컴퓨터의 주류가 MIMD 모델로 가고 있다는 것은 거의 확실시 되고 있다. 그러나 MIMD를 구성하고 있는 각 프로세서의 설계를 주도하는 기술적, 시장적 주된 요소는, 최소한 현재까지는, 대규모 병렬처리 컴퓨터가 아니라 개인용 컴퓨터나 워크스테이션과 같은 단일 프로세서 시스템이다. 이는 이들 고성능 마이크로프로세서들이 대규모 병렬처리에 적합한 요소들을 이상적으로 갖추지 못하고 있음을 의미한다. 무엇보다도 이들 상용 고성능 마이크로프로세서들은 노드간 통신과 동기화 지원 기능이 부족하다.

프로세서 내부의 레지스터 단계에서 네트워크와 접속함으로써, 통신 오버헤드를 줄인 몇몇 연구(J-machine MDP[23], Monsoon[24], EM-4[20])가 있지만, 이들이 상용화되려면 기존의 상용 마이크로프로세서 보다 우수한 성능을 내야 하고 호환성 문제를 극복해야 하는 어려움이 있다. 한편, 현재 상용화된 마이크로프로세서를 이용하여 시스템을 구성하는 경우 노드당 최대 성능은 우수하지만, 노드간 통신 오버헤드와 동기화를 효과적으로 통제하지 못하면 좋은 전체 성능을 기대할 수 없다.

메시지 전달 방식은 주소 공간의 분할과 복잡한 자료 구조 전달의 어려움 등으로 인해 프로그래밍이 쉽지 않다. 따라서 많은 컴퓨터들이 확장성을 위해 물리적으로는 분산 메모리를 갖고 있음에도 불구하고, 이들 컴퓨터에서 사용자에게 전역적인 가상 공유 메모리 공간을 제공하려는 노력을 기울이고 있다. 이미 살펴본 바와 같이 여기에는 전역 캐쉬를 제공함으로써 공유 메모리를 제공하는 방법과 메시지 전달에 기초한 방법이 있다.

대규모 병렬처리 컴퓨터에서 가장 중요한 것은 아마도 확장성일 것이다. 확장성은 다양한 요소에 의해 지배되지만, 주로 프로세서와 메모리의 연결 형태에 의해 결정된다. 현재까지는 [프로세서-메모리]-연결망 형태를 갖는 분산 메모리 형태가 확장성에 유리한 것으로 여겨진다. 또 [프로세서-메모리]로 된 노드들간의 연결 위상들(링, 메쉬, 팻 트리, 하이퍼큐브, 큐브 연결 사이클 등)도 확장성과 시스템의 특성에 영향을 미치는 중요한 요소 중의 하나이다.

참고문헌

[1] H. Burkhardt III et al., "Overview of the KSR1 Computer System," *Technical Report,*

KSR-TR-9202001, Kendall Square Research, Boston, MA, 1992.

[2] IEEE Standard for Scalable Coherent Interface, IEEE Std 1596-1992, 1993.

[3] D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 148-159, 1990.

[4] D. Lenoski et al., "The DASH Prototype : Implementation and Performance," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 92-103, 1992.

[5] T. M. Corp., "The Connection Machine Cm-5 Technical Summary," *Technical Report*, Thinking Machines Corp., 1992.

[6] C. E. Leiserson et al., "The Network Architecture of the Connection Machine CM-5," *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[7] Bob Duzett and Ron Buck, "An Overview of the nCUBE 3 Supercomputer," *The 4th Symposium on the Frontiers of Massively Parallel Computation*, pp. 458-464, 1992.

[8] R. H. Saavedra-Barrera, D. E. Culler, T. von Eicken, "Analysis of Multithreaded Architectures for Parallel Computing," *Proceedings of 2nd ACM Symposium on Parallel Algorithms and Architectures*, 1990.

[9] G. M. Papadopoulos et al., "*T : Integrated Building Blocks for Parallel Computing," *MIT* CSG Memo 351, 1993.

[10] Intel Corp., *Paragon XP/S Product Overview*, 1991.

[11] T. von Eicken et al., "Active Messages : a Mechanism for Integrated Communication and Computation," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[12] Allan Gottlieb et al., "NYU Ultracomputer," *IEEE Trans. on Computers*, 1983.

[13] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor System," *Real-time Signal Processing IV*, Vol. 298, pp. 241-248, 1981.

[14] Arvind, R. S. Nikhil and K. K. Pingali, "I-Structures : Data Structures for Parallel Computing," *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 4, 1989.

[15] R. S. Nikhil, G. M. Papadopoulos and Arvind, "*T : A Multithreaded Massively Parallel Architecture," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[16] R. E. Kessler and J. L. Schwarzmeier, "Cray T3D : A New Dimension for Cray Research," *CompCon93*, pp. 176-182, 1993.

[17] A. Agarwal et al., "The MIT Alewife Machine : A Large-Scale Distributed-Memory Multiprocessor," *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[18] J. Archibald and J. L. Baer, "Cache Coherent Protocols : Evaluation Using a Multirpocessor Simulation Model," *ACM Trans. on Computer Systems*, 1986.

[19] C. E. Leiserson, "Fat Trees : Universal Network for Hardware-Efficient Supercomputing," *IEEE Trans. on Computers*, Vol. C-34, No. 10, pp. 892-901, 1985.

[20] S. Sakai et al., "An Architecture of a Dataflow Single Chip Processor," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 46-53, 1989.

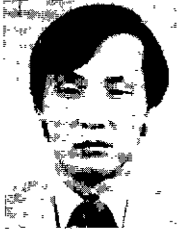
[21] J. Kuskin et al., "The Stanford FLASH Multiprocessor," *Proceedings of the 21th Annual International Symposium on Computer Architecture*, 1994.

[22] Gordon Bell, "Ultracomputers : A Teraflop Before Its Time," *Communication of the ACM*, Vol. 35, No. 8, 1992.

[23] W. Dally et al., "Architecture of a Message-Driven Processor," *Proceedings of 14th Annual Symposium on Computer Architecture*, pp. 189-196, 1987.

[24] G. M. Papadopoulos and D. E. Culler, "Monsoon : An Explicit Token Store Architecture," *Proceedings of 17th Annual Symposium on Computer Architecture*, 1990.

한 상 영



1972 서울대학교 공과대학 응용수학과 공학과
1977 서울대학교 자연과학대학 계산통계학과 이학석사
1977~1978 울산 공과대학 전임강사
1983 미국 Austin소재 Texas 대학교 이학박사
1983~1984 미국 Austin소재 Texas 대학교 연구원
1984~1988 서울대학교 자연과학대학 계산통계학과 조교수
1988~1993 서울대학교 자연과학대학 계산통계학과 부교수
1993 4~현재 서울대학교 자연과학대학 계산통계학과 교수
관심분야 : 프로그래밍 언어, 병렬처리

최 영 근

1980 서울대학교 사범대학 수학 교육과 이학사
1982 서울대학교 계산 통계학과 이학 석사
1989 서울대학교 계산 통계학과 이학 박사
1983~1994 광운대학교 이과대학 전자계산학과 부교수
1995 광운대학교 이과대학 전자계산학과 교수, 학과장
1990~현재 한국 정보과학회 병렬처리 시스템 연구회 운영위원
관심분야 : 병렬 컴파일러, 병렬 프로그래밍 언어, 프로그래밍 검증 등

원 영 주

1981 육군사관학교 전자공학과
1987 미네소타대 전산학 석·박사
1989 정보과학회 병렬처리시스템 연구회 학술 및 편집간사
현재 육군사관학교 전산학과 부교수
연구분야 : 병렬처리 알고리즘, 병렬처리 컴퓨터 구조