데이타 병렬 프로그램에서 루프 세부 분할 및 동적 스케쥴링을 통한 통신과 계산의 중첩 모델

(A Communication and Computation Overlapping Model through Loop Sub-partitioning and Dynamic Scheduling in Data Parallel Programs)

김 정 환 † 한 상 영 †† 조 승 호 ††† 김 흥 환 ††††
(Junghwan Kim) (Sangyong Han) (Seungho Cho) (Heunghwan Kim)

요 약 본 논문은 데이타 병렬 프로그램에서 효율적 통신을 위한 하나의 방법으로 통신과 계산 작업을 중첩하여 실행하는 모델을 제안한다. 이 중첩 모델에서는 통신 지연 시간 동안 중첩하여 수행할 계산 작업을 얻기 위해 주어진 루프 분할을 다시 세부 분할한다. 주어진 루프 분할은 다른 외부 데이타 분할을 참조하기도 하지만, 루프 분할의 모든 반복들이 항상 외부 데이타 참조를 필요로 하는 것은 아니다. 따라서 주어진 루프 분할을 외부 데이타를 요구하는 루프 반복들의 집합과 그렇지 않은 루프 반복들의 집합으로 나눌 수 있다. 이렇게 나누어진 루프 세부 분할은 효율적인 수행을 위해 메시지 도착 순서에 따라 동적으로 스케쥴링된다. 제안된 방법에 따라 IBM SP2에서 몇가지 프로그램으로 실험을 한 결과, 중첩 모델이 성능 향상을 보임을 확인할 수 있었다.

Abstract We propose a model which overlaps communication with computation for efficient communication in the data-parallel programming paradigm. The overlapping model divides a given loop partition into several sub-partitions to obtain computation which can be overlapped with communication. A loop partition sometimes refers to other data partitions, but not all iterations in the loop partition require non-local data. So, a loop partition may be divided into a set of loop iterations which require non-local data, and a set of loop iterations which do not. Each loop sub-partition is dynamically scheduled depending on associated message arrival. The experimental results for a few benchmarks in IBM SP2 show enhanced performance in our overlapping model.

1. 서 론

분산 메모리 다중컴퓨터는 공유 메모리 다중프로세서에 비해 확장성이 뛰어나고, 비교적 적은 비용으로 높은 성능을 얻을 수 있다. 가령, IBM SP2, Intel Paragon, Thinking Machines CM-5와 같은 분산 메모리 다중컴퓨터는 수백개의 노드로 구성될 수 있으며, 대규모 병렬성을 지원한다.

그런데, 분산 메모리 다중컴퓨터의 하위 환경에서는 프로그래밍하기가 쉽지 않으며, 효율적인 프로그램을 작성하기 위해서는 매우 까다로운 작업을 해야 한다. 이런 어려움의 주된 이유는 단일의 전역 주소 공간(global address space)을 제공하지 못하기 때문이다. 이것은 프로그래머로 하여금 수작업으로 계산과 데이타를 각 노드에 분산시키고, 또한 통신을 명시적으로 관리하도록 요구한다.

† 비 회 원 : 삼성전자 연구원
automata@pplab.snu.ac.kr
†† 종신회원 : 서울대학교 전자과학과 교수
syhan@pplab.snu.ac.kr
††† 비 회 원 : 강남대학교 산업컴퓨터전자공학부 교수
shcho@kns.kangnam.ac.kr
†††† 비 회 원 : 건국대학교 전산과학과 교수
khh@kcucc.cj.konkuk.ac.kr
논문접수 : 1998년 11월 20일
심사완료 : 1999년 10월 1일

따라서 많은 연구들이 프로그래머의 이러한 부담을 덜어주는데 집중되었다. Fortran D[1], Fortran 90D[2], Vienna Fortran[3], High Performance Fortran[4]과 같은 데이타 병렬 언어들은 프로그래머에 의한 데이타 분산의 명시를 요구하지만, 명시된 데이타 분산 정보를 이용하여 계산 분할을 자동적으로 수행하며, 또한 필요한 메시지 송수신 등의 통신을 합성한다. 데이타 분산의 명시를 제외하면, 이러한 언어들은 프로그래머에게 공유 메모리 형태의 프로그래밍 모델을 제공한다.

한편, 분산 메모리 다중컴퓨터에서의 통신은 상당히 긴 지연 시간이 소요되기 때문에, 메시지의 송수신 단계는 큰 부담이 된다. 따라서 적절한 데이타 분할을 통해 통신량을 최소화하는 것은 매우 중요한 일이다. 그렇지만, 일단 통신이 불가피한 경우라면, 이를 효율적으로 처리하는 것도 중요하다.

본 논문에서는 데이타 병렬 프로그램에서의 통신 부담을 가급적 줄이기 위해 통신 시간 동안 다른 계산 작업을 중첩하여 실행하는 한가지 모델을 제안한다. 통신과 중첩하여 실행할 계산 작업은 통신에 비종속적인 부분이어야 한다. 이러한 코드 부분을 프로그램 전체에서 탐색하여 얻을 수도 있지만, 명령형 언어에서 이러한 코드 부분을 찾기는 쉽지 않다. 본 논문에서는 통신을 기다리는 루프 자체를 실제로 통신을 필요로 하는 부분과 그렇지 않은 부분으로 분할함으로써 이 문제를 해결할 것이다.

2. 기존 연구

통신 최적화는 크게 통신 오버헤드를 줄이는 것과 통신 지연 시간을 감추는 것으로 나눌 수 있다. 통신 오버헤드를 줄이기 위한 방법은 메시지 벡터화(message vectorization) [6][7], 메시지 합착(message coalescing) [5], 메시지 집합화(message aggregation) [1], 군집 통신(collective communication)[8] [9] 등이다. 통신 지연 시간을 감추기 위한 방법으로는 메시지 파이프라이닝(message pipelining), 벡터 메시지 파이프라이닝(vector message pipelining), 반복 재순서화(iteration reordering) 등이 있는데, 모두 통신과 계산의 중첩을 이용하고 있다.

통신 시간은 T_{start}, $T_{copy}(n)$, $T_{transit}(n)$의 세 부분으로 나누어 생각할 수 있다. 먼저, T_{start}는 송신 또는 수신에 기본적으로 소요되는 시간으로 메시지 길이와 무관하다. $T_{copy}(n)$은 길이 n의 메시지를 전송할 때, 사용자 주소 공간에서 시스템 버퍼로 복사하는데 걸리는 시간이다. $T_{transit}(n)$은 길이 n의 메시지가 양극단 사이를 이동할 때 걸리는 시간이다. T_{start}는 비교적 긴 시간이기 때문에, 메시지의 길이가 매우 길 때를 제외하면 전체 시간에서 차지하는 비중이 높다. 가령, Intel iPSC는 1바이트를 전송할 때 걸리는 시간이 약 95 μsec이지만, 그 다음 1바이트를 더 전송하는데 걸리는 시간은 0.4 μsec에 불과하다[8]. 또한 IBM SP2는 1바이트 전송에 약 40 μsec, 그 다음 1바이트 전송에 약 0.028 μsec이 소요된다[10].

메시지 벡터화는 데이타 종속성 분석을 통해 단위 요소 별의 메시지들을 하나의 벡터 메시지로 결합한다. 따라서 메시지의 수를 줄일 수 있기 때문에, 전체적으로 T_{start} 시간을 줄일 수 있다. 하나의 배열 참조 인덱스에 대해 메시지 벡터화가 진행되고 나면, 메시지 합착을 통해 전체 참조 인덱스들에 대해 하나의 벡터로 합치게 된다. 메시지 합착은 각 데이타 요소가 하나의 프로세서에 한번만 보내지는 것을 보장한다. 한편, 메시지 집합화는 각 프로세서에 단 하나의 메시지가 보내지는 것을 보장한다. 메시지 집합화에서 하나의 목적지에 대한 모든 메시지 벡터화와 합착의 결과는 하나로 모아진다. 따라서 부가적으로 버퍼 복사가 수반된다. 메시지 벡터화, 합착, 집합화는 모두 메시지 수를 줄이기 때문에 T_{start} 시간을 감소시킨다.

메시지 파이프라이닝은 각 비지역 참조(nonlocal reference)에 대해, send 명령은 해당 데이타가 정의된 후 가능한 한 바로 이슈되고 receive 명령은 가능한 한 그 값이 사용되기 바로 직전에 이슈되도록 코드 배치를 하는 방법이다[11]. 이 방법은 데이타의 정의(definition)와 사용(use) 사이에 다른 계산을 수행하도록 함으로써 $T_{transit}$ 시간을 감춘다. 그러나, 이 방법은 각 단위 참조별로 파이프라이닝을 시도함으로써, 메시지 벡터화와 같은 다른 최적화를 막는다. 메시지 벡터화와 같이 T_{start} 시간을 줄이는 것은 매우 큰 역할을 하기 때문에, 이 방법을 통해 $T_{transit}$ 시간을 감추는 것은 오히려 엄청난 통신 비용의 증가를 가져올 수 있다. 따라서 제한적으로 신중히 적용되어야 할 기법이다.

벡터 메시지 파이프라이닝은 전체 통신 비용을 증가시키지 않고 $T_{transit}$ 시간을 감춘다. 이것은 일단 메시지 벡터화가 진행된 후에, 각 벡터 메시지의 send와 receive를 메시지 파이프라이닝과 유사한 방법에 따라 배치한다. 한편, 벡터 메시지 파이프라이닝은 실제 구현에 있어 여러가지 다른 요소에 의해 제약될 수 있다. 가령, 프로그램에 내재되어 있는 데이타 종속성은 벡터 send나 벡터 receive의 이동을 제한한다.

반복 재순서화 역시 $T_{transit}$ 시간을 감추기 위한 방법

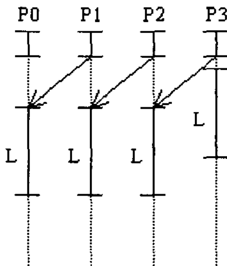
이다. 이 방법은 원래의 루프 반복들을 지역적으로 실행 가능한 루프 반복과 통신이 필요한 루프 반복으로 나눈 후, send와 receive 명령 사이에 지역적으로 실행 가능한 루프 반복을 배치함으로써, 통신 시간을 감춘다[12]. 벡터 메시지 파이프라이닝은 데이타 종속성 등으로 인해 적용될 수 없는 경우가 있는데, 이 방법은 그러한 경우라도 단일의 루프 구조에 대해 통신 시간을 감출 수 있는 방법을 제공한다. 그러나, 다차원에 대한 반복 재순서화는 그 분할과 루프 경계를 다루는 것이 쉬운 문제는 아니다. 또한 분할된 루프 반복 집합들의 실행 순서를 정적으로 정하는 것도 어려운 문제이다. 본 논문에서는 이러한 다차원 분할과 루프 반복 집합들의 동적 스케줄링에 관한 문제를 다룰 것이다. 한편, 반복 재순서화는 루프 반복 집합의 세분화로 인한 캐쉬 성능의 저하 등과 같은 역효과를 가져올 수 있다[5].

3. 통신과 계산의 중첩 실행

본 논문에서 제안하는 통신과 계산의 중첩 실행 모델은 크게 두가지 방법에 기초하고 있다. 하나는 루프 세부 분할이고 다른 하나는 동적 스케줄링이다. 통신과 중첩 실행할 수 있는 계산 작업은 통신 결과를 이용하지 않는 작업들이어야 한다. 이러한 독립적인 계산 작업을 본 논문에서는 루프 세부 분할을 통해 얻을 것이다.

```
real A(100, 26), B(100, 26)
if (my$p .gt. 0) send(B(1:100, 1), my$p-1)
if (my$p .lt. 3) recv(B(1:100, 26), my$p+1)
do j = 1, 25
  do i = 1, 100
     A(i, j) = B(i, j+1)                L
  enddo
enddo
```

(a)



(b)

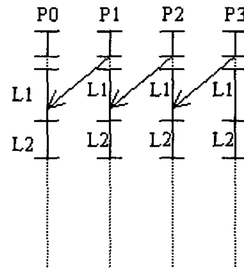
그림 1 통신 / 계산 중첩을 하지 않았을 때

그림 1의 (a)는 외부 데이타 참조가 루프 반복 집합의 경계에서만 발생한다. 그럼에도 불구하고, 루프 전체가 통신의 종료를 기다려야 하는 이유는 메시지 벡터화를 통해 통신 코드가 루프 밖으로 이동했기 때문이다.

외부 데이타 참조는 루프 반복 집합 (25, 1:100)에서만 발생하고, 나머지 (1:24, 1:100)은 내부 데이타 참조만 일어나므로 루프 반복 집합을 두개의 부분으로 나눌 수 있다. 여기서 (25, 1:100)은 j=1, i=1~100인 루프 반복들을 의미한다. 앞으로 (l_1:u_1, l_2:u_2)는 첫 번째 루프의 제어 변수가 l_1~u_1이고, 두 번째 루프의 제어 변수가 l_2~u_2인 루프 반복 집합을 나타내기로 한다. 루프 반복 집합 (1:24, 1:100)은 메시지 send와 receive 사이에 수행하도록 함으로써, 통신과 계산의 중첩 효과를 얻을 수 있다. 그림 2의 (a)는 통신과 계산이 중첩 수행되도록 루프 반복 집합을 분할한 프로그램이다. (b)는 메시지 송신이 끝나자마자 바로 지역적으로 수행할 수 있는 부분을 수행하고, 메시지가 도착하면 이에 따라 나머지 부분을 수행할 수 있음을 보여준다.

```
real A(100, 26), B(100, 26)
if (my$p .gt. 0) send(B(1:100, 1), my$p-1)
do j = 1, 24
  do i = 1, 100
     A(i, j) = B(i,j+1)                 L1
  enddo
enddo
if (my$p .lt. 3) recv(B(1:100, 26), my$p+1)
do j = 25, 25
  do i = 1, 100
     A(i, j) = B(i, j+1)                L2
  enddo
enddo
```

(a)



(b)

그림 2 통신 / 계산 중첩 실행

앞으로 내부 데이타 참조만 있는 루프를 **지역 루프**, 외부 데이타 참조가 포함되어 있는 루프를 **비지역 루프**라 부르기로 하자. 앞의 그림 2와 같이 지역 루프와 단일의 비지역 루프를 갖는 경우는 컴파일 시에 정적으로 스케줄링할 수 있지만, 만일 비지역 루프가 여러 개가

있다면, 이들 간의 동적인 스케줄링이 고려되어야 한다.

4. 루프 세부 분할 방법

이 장에서는 루프 세부 분할 방법에 대해 구체적으로 논의될 것이다. 루프 세부 분할은 배열 인덱스 수식의 분석을 통해 이루어진다.

4.1 1차원 분할

데이타 분할이 1개 차원에 대해서만 이루어져 있을 때, 인접 데이타 분할은 최대 2개이다. 여기서 각 인접 데이타 분할의 참조 여부를 0/1로 표시했을 때, 가능한 조합의 수는 {(0, 0), (0, 1), (1, 0), (1, 1)}의 모두 2^2 = 4가지이므로, 루프 세부 분할은 최대 4개까지 가능하다. 이중에서 (0, 0)은 어느 인접 데이타 분할도 참조하지 않는, 즉 지역 루프의 경우이고, 나머지 3개는 비지역 루프의 경우이다. 그런데, 양쪽 인접 데이타 분할을 모두 참조하는 비지역 루프 (1, 1)이 존재한다면, 지역 루프는 존재하지 않는다. 반대로 지역 루프가 존재하면, 비지역 루프 (1, 1)은 존재하지 않는다. 따라서 실제로는 3개의 세부 분할만이 존재한다.

인접 데이타 분할을 참조하는 배열 인덱스의 수식은 i+c 형태를 갖는다. 그림 3의 (a)는 지역 루프가 존재하는 경우, (b)는 지역 루프가 존재하지 않는 경우에 대한 예이다. 분할의 크기가 10이라고 할 때, (a)는 루프 내의 문장이 A(i) = B(i+3)+B(i-2) 인 경우로써, 2개의 비지역 루프와 1개의 지역 루프로 분할된다. 반면 (b)는 A(i) = B(i+7)+B(i-6)인 경우로써, 3개의 비지역 루프로 분할된다. 이중 1개는 양쪽 데이타 분할을 모두 참조하는 비지역 루프이다.

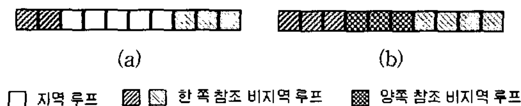


그림 3 1차원에서의 계산 세부 분할 예

i+c 형태에서 |c|는 보통 데이타 분할의 크기 Ds보다 작으므로 인접 데이타 분할 바깥 쪽의 분할들을 참조하는 일은 발생하지 않는다. |c| > Ds인 경우는 지나치게 분할을 작게 한 것으로 최적의 데이타 분할이 아니다. 또한 대개 |c|는 Ds보다 충분히 작으므로 (a) 경우가 대부분일 것이다.

이제 여러 개의 배열 인덱스 수식 $i+c_1$, $i+c_2$, ..., $i+c_k$이 주어졌을 때 계산 세부 분할을 얻는 방법을 살펴보자. 앞의 예에서 알 수 있듯이 c_i의 양수, 음수 여부에 따라 어느 쪽 인접 데이타 분할을 참조하는지가 결정된다. 각각의 루프 세부 분할의 크기는 $|c_i|$들 중 가장 큰 값에 의해 결정된다. 따라서 양수, 음수에서 c_i들의 최대 절대값 Mc, mc을 구해보면 다음과 같다.

$$Mc = \max(\{c_i \mid c_i \geq 0\} \cup \{0\})$$
$$mc = |\min(\{c_i \mid c_i < 0\} \cup \{0\})|$$

예를 들어 그림 3에서는 (a)의 경우 Mc = 3, mc = 2이고, (b)의 경우 Mc = 7, mc = 6이 된다. 이제, 주어진 루프 분할이 (l : u)라고 할 때, 세부 분할은 다음과 같이 구할 수 있다.

(Mc + mc) < Ds 인 경우:
 세부 분할 (0, 1) = ((u - Mc + 1) : u)
 세부 분할 (1, 0) = (l : (l + mc - 1))
 세부 분할 (0, 0) = ((l + mc) : (u - Mc))
(Mc + mc) ≥ Ds 인 경우:
 세부 분할 (0, 1) = ((l + mc) : u)
 세부 분할 (1, 0) = (l : (u - Mc))
 세부 분할 (1, 1) = ((u - Mc + 1) : (l + mc - 1))

그림 3의 경우 세부 분할을 계산해보면, (a)는 (0, 1) = (8 : 10), (1, 0) = (1 : 2), (0, 0) = (3 : 7)이 되고, (b)는 (0, 1) = (7 : 10), (1, 0) = (1 : 3), (1, 1) = (4 : 6)이 된다.

4.2 2차원 분할

2차원 데이타 분할에서 정확한 루프 세부 분할을 사용하는 것은 비효율적일 수 있다. 그 이유는 먼저 2차원 데이타 분할에서 인접 데이타 분할은 모두 8개이고, 이들 인접 데이타 분할에 대한 참조 여부에 대한 경우의 수는 2^8=64 가지로 매우 많다는 사실이다. 앞서 1차원 분할의 경우와 마찬가지로 이들 경우의 수가 모두 존재하는 것은 아니지만, 지나치게 세분화된 분할은 루프 제어 오버헤드와 캐쉬 효과의 감소 등으로 오히려 성능이 나빠질 수 있다. 따라서, 본 논문에서는 루프 세부 분할의 근사(approximation)를 사용한다.

2차원 분할에서 근사 후의 계산 세부 분할은 그림 4와 같이 항상 최대 9개까지만 갖게 된다. 근사에서 루프 세부 분할 L8은 외형상 데이타 분할 A, B, C를 모두 참조하는 루프 세부 분할이지만, 실제로는 L8의 일부인 어떤 루프 반복들은 A, B만을 또는 B, C만을 참조할 수 있다. 그러나 이러한 루프 반복별로 세부 분할을 구성할 경우 루프 제어가 복잡해질 뿐 아니라, 매우 작은 수의 반복으로 구성된 루프들이 만들어지므로 상대적으로 루프 제어 오버헤드가 차지하는 비중이 커진다. 또한 시간 지역성이 적어지므로 캐쉬의 효과도 감소된다.

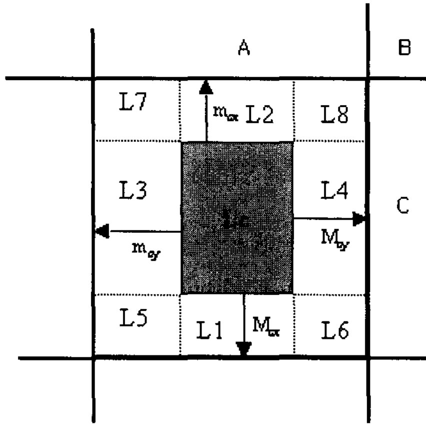


그림 4 2차원 데이타 분할에서 계산 세부 분할

이제 비지역 루프 L1 ~ L8 및 지역 루프 Lc를 구하는 방법을 살펴보면 다음과 같다. 루프 내의 문장에서 사용되는 배열 인덱스들의 집합이 다음과 같이 표현된다. 배열 인덱스 수식 $(i+c_x, j+c_y)$ 에서 오프셋의 절대값 $|c_x|$ 및 $|c_y|$는 데이타 분할의 세로(Ds_x), 가로(Ds_y)보다 각각 충분히 작다는 것을 가정한다.

$$\{ (i+c_{x1}, j+c_{y1}), (i+c_{x2}, j+c_{y2}), ..., (i+c_{xk}, j+c_{yk}) \}$$

오프셋 쌍 만을 추출한 집합을 **오프셋 집합** 또는 **스텐실**(stencil)이라고 부르기로 하며, 그 집합은 다음과 같다.

$$\{ (c_{x1}, c_{y1}), (c_{x2}, c_{y2}), ..., (c_{xk}, c_{yk}) \}$$

1차원의 경우에서처럼 양수 및 음수 별로 가장 큰 절대값의 오프셋이 세부 분할의 크기를 결정한다. 차원 간의 연관성을 배제하고, Mc_x, mc_x, Mc_y, mc_y를 구해보면 다음과 같다.

$$Mc_x = \max(\{c_{xi} \mid c_{xi} \geq 0\} \cup \{0\})$$
$$mc_x = |\min(\{c_{xi} \mid c_{xi} < 0\} \cup \{0\})|$$
$$Mc_y = \max(\{c_{yi} \mid c_{yi} \geq 0\} \cup \{0\})$$
$$mc_y = |\min(\{c_{yi} \mid c_{yi} < 0\} \cup \{0\})|$$

루프 반복 집합 분할이 $(l_x : u_x, l_y : u_y)$일 때, 각 세부 분할은 다음과 같다.

$$L1 = ((u_x - Mc_x + 1) : u_x, (l_y + mc_y) : (u_y - Mc_y))$$
$$L2 = (l_x : (l_x + mc_x - 1), (l_y + mc_y) : (u_y - Mc_y))$$
$$L3 = ((l_x + mc_x) : (u_x - Mc_x), l_y : (l_y + mc_y - 1))$$
$$L4 = ((l_x + mc_x) : (u_x - Mc_x), (u_y - Mc_y + 1) : u_y)$$
$$L5 = ((u_x - Mc_x + 1) : u_x, l_y : (l_y + mc_y - 1))$$
$$L6 = ((u_x - Mc_x + 1) : u_x, (u_y - Mc_y + 1) : u_y)$$
$$L7 = (l_x : (l_x + mc_x - 1), l_y : (l_y + mc_y - 1))$$
$$L8 = (l_x : (l_x + mc_x - 1), (u_y - Mc_y + 1) : u_y)$$
$$Lc = ((l_x + mc_x) : (u_x - Mc_x), (l_y + mc_y) : (u_y - Mc_y))$$

만일 구한 세부 분할의 어떤 차원이 $lb > ub$ 인 경우에는 그 세부 분할은 존재하지 않는다. 이 경우 전체 세부 분할 개수는 9개보다 적어진다.

4.3 다차원 분할

이제 n차원 데이타 분할에서 루프 세부 분할을 얻는 방법을 살펴보자. 2차원의 경우와 마찬가지로 인덱스 수식에서 각 차원의 오프셋은 데이타 분할 크기에 비해 충분히 작다는 것을 가정한다. 오프셋 집합은 다음과 같이 나타낸다.

$$\{ (c_{11}, c_{21}, ..., c_{n1}), (c_{12}, c_{22}, ..., c_{n2}), ..., (c_{1k}, c_{2k}, ..., c_{nk}) \}$$

i번째 차원에 대한 Mc_i 및 mc_i는 다음과 같다.

$$Mc_i = \max(\{c_{ij} \mid c_{ij} \;\; 0\} \; \{0\})$$
$$mc_i = |\min(\{c_{ij} \mid c_{ij} < 0\} \; \{0\})|$$

각 세부 분할을 나타내기 위해 $L(e_1, e_2, ..., e_n)$ 기호를 사용하기로 한다. 여기서 e_i는 {-1, 0, 1} 중의 한 값을 가지는데, -1인 경우는 i차원에서 좌측(음의 방향)에 있는 인접 분할을 참조하며, 1인 경우는 우측(양의 방향)에 있는 인접 분할을 참조하고, 그리고 0인 경우는 지역 데이타만을 참조한다. 2개 이상의 차원이 -1 또는 1이면, 그 두 인접 분할 사이에 있는 또 다른 인접 분할도 참조할 가능성이 있다.

이제 n차원에서의 루프 분할이 ($l_1 : u_1, l_2 : u_2, ..., l_n : u_n$)일 때, 루프 세부 분할 $L(e_1, e_2, ..., e_n)$은 다음과 같다.

$$L(e_1, e_2, ..., e_n) = (lb_1 : ub_1, lb_2 : ub_2, ..., lb_n : ub_n)$$

e_i = 1일 때:
$$lb_i = (u_i - Mc_i + 1)$$
$$ub_i = u_i$$
e_i = -1일 때:
$$lb_i = l_i$$
$$ub_i = (l_i + mc_i - 1)$$
e_i = 0일 때:
$$lb_i = (l_i + mc_i)$$
$$ub_i = (u_i - Mc_i)$$

5. 동적 스케쥴링

각각의 비지역 루프는 종속 관계에 있는 메시지 수신이 존재한다. 왜냐하면, 비지역 루프의 정의 자체에 외부 데이타 참조가 있음을 내포하기 때문이다. 따라서 메

시지 수신 코드와, 이것과 종속 관계에 있는 비지역 루프는 분리되어 스케쥴링될 수 없다. 이제 메시지 수신 코드 R_i와 이와 연결된 비지역 루프 L_i를 $L_i \rightarrow R_i$로 나타내기로 한다. 하나의 메시지 수신 코드 R_i에는 여러 개의 비지역 루프가 종속되어 있을 수 있다. 반대로 하나의 비지역 루프가 여러 개의 메시지 수신 코드에 종속될 수 있다. 그림 5는 메시지 수신 코드와 비지역 루프간의 다양한 종속 그래프를 보여준다.

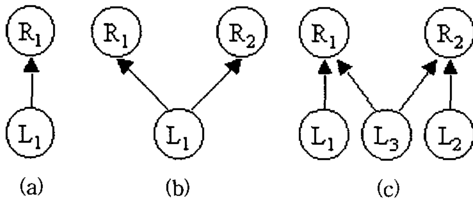


그림 5 메시지 수신과 비지역 루프 간의 종속 그래프

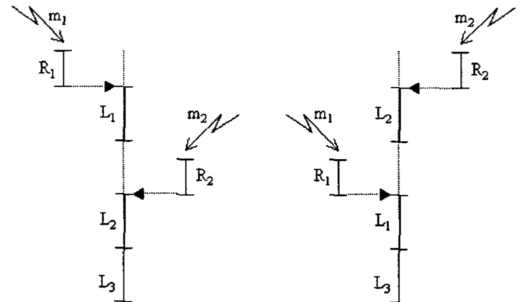
각각의 종속 관계는 코드 실행의 선행 관계를 나타낸다. 그래프에서 비지역 루프 L_i가 메시지 수신 코드 R_j에 종속되어 있다면, R_j는 반드시 L_i보다 먼저 수행되어야 함을 나타낸다. 그러나, 메시지 수신 R_i와 R_j 간에는 종속 연결이 없으므로 이들 간의 실행 순서는 부여되지 않는다. 따라서 이들은 부분 순서화 집합(partially ordered set)을 형성한다. 이제 이 부분 순서화 집합을 컴파일시에 정적으로 스케쥴링하는 것을 고려해보자. 그림 5의 (c) 경우에서 다양한 스케쥴링이 존재하지만, 그 중의 하나는 다음과 같다.

$$R_1 - L_1 - R_2 - L_2 - L_3$$

이처럼 정적으로 스케쥴링되어 생성된 코드는 실행 시간 부담이 없지만, 동적인 통신 상황에 효과적이지 못하다. 가령, R_1에 의해 수신될 메시지가 먼저 도착하지 않고, R_2에 의해 수신될 메시지가 먼저 도착한다면, 위 스케쥴링은 최적의 해가 아니다.

본 논문에서는 메시지 도착에 의해 구동되는 동적인 스케쥴링 방법을 제안한다. 이 방법에서는 그림 6과 같이 먼저 도착하는 메시지에 따라 R_1 또는 R_2가 활성화되어 스케쥴링된다. 다차원의 데이타 분할에서는 비지역 루프도 그림 5보다 훨씬 많은 경우가 생긴다. 이러한 경우 메시지 구동 스케쥴링(message-driven scheduling)이 다양한 동적 상황에 유리하다.

한편, 메시지 수신 R_i에는 호응하는 메시지 송신 S_i가 존재하는데, 메시지 송신 코드 S_1, S_2, ..., S_k는 정적 스케쥴링을 통해 생성한다. 이때 S_i와 R_i는 서로 같은 태그를 공유해야 한다. 그런데, S_i에 의해 송신된 메시지가 도착하면, 수신측 노드의 스케쥴러는 R_i를 스케쥴링해야 하므로 메시지의 태그에는 스케쥴링할 대상인 R_i에 대한 정보가 내포되어야 한다. 이러한 정보는 R_1, ..., R_k에 대한 식별자일 수도 있고, Active Message [13]와 같이 메시지 수신 처리기(handler) 역할을 하는 R_i에 대한 주소일 수도 있다.



(a) m_1이 먼저 도착하는 경우 (b) m_2가 먼저 도착하는 경우

그림 6 메시지 구동 스케쥴링

그림 7의 (a)는 루프 경계의 양쪽에서 외부 데이타 참조가 발생하는 프로그램 예이다. 이 프로그램을 통신과 계산의 중첩 실행 모델에 따라 변환하면, (b)와 같은 프로그램이 된다.

그림 7의 (b)에서 next는 하나의 스케쥴링 단위가 끝났을 때, 스케쥴러로 제어를 이동시키기 위한 명령이다. 지역 루프의 실행이 끝나고 next 문을 만나면, 제어는 스케쥴러로 이동하여 메시지의 도착을 기다린다. 메시지 도착 순서에 따라 [R_1-L_1] 또는 [R_2-L_2] 중 먼저 실행된 코드는 join 문장에서 나머지 하나의 실행이 끝나기를 기다린다. 프로세서 0과 프로세서 3은 메시지 수신이 1개만 이루어지므로, join 문장에 도달하는 스레드는 단지 1개 뿐이다. 따라서 fork 문에 의해 인위적으로 join 문에 도달하는 스레드를 1개 더 만든다.

스케쥴러는 실행 시간 코드로 주어지는데, 그 구현 방법은 앞서 언급한 것처럼 두가지 방법이 가능하다. 그림 7의 경우에서 Active Message에 의해 구현된다면, 각 메시지는 R_1, R_2에 대한 시작 주소를 헤더에 포함할 것이다.

6. 성능 평가

6.1 실험 환경

실험에 사용한 시스템은 IBM SP2로서 분산 메모리

```
real A(100, 100), B(100, 100)
parameter (n$proc = 4)
decomposition D(100, 100)
align A, B with D
distribute D(:, BLOCK)
do j = 2, 99
    do i = 1, 100
        A(i, j) = B(i, j+1) + B(i, j-1)
    enddo
enddo
```

(a) 원시 프로그램

```
        real A(100, 0:26), B(100, 0:26)
        if (my$p .lt. 3) send(R1, B(1:100, 25), my$p+1)
        if (my$p .gt. 0) send(R2, B(1:100, 1), my$p-1)
        do j = 2, 24
            do i = 1, 100
                A(i, j) = B(i, j+1) + B(i, j-1)
            enddo
        enddo
        if (my$p .eq. 0) fork JJ
        if (my$p .eq. 3) fork JJ
        next
R1      recv(B(1:100, 0), my$p-1)
L1      do j = 1, 1
            do i = 1, 100
                A(i, j) = B(i, j+1) + B(i, j-1)
            enddo
        enddo
        goto JJ
R2      recv(B(1:100, 26), my$p+1)
L2      do j = 25, 25
            do i = 1, 100
                A(i, j) = B(i, j+1) + B(i, j-1)
            enddo
        enddo
JJ      join(jc, 2)
```

(b) 통신/계산 중첩 모델에 따라 변환된 프로그램

그림 7 통신/계산 중첩 예

를 갖는 다중컴퓨터 시스템이다. 일반적으로 2~128개의 노드를 갖지만, 512개까지의 큰 시스템 구성도 가능하다. SP2의 각 노드는 POWER2 프로세서의 RISCSystem/6000이 사용되며, 이들 노드들은 HPS(High Performance Switch)[14]라고 불리는 다단계 패킷 교환망(multistage packet-switched network)에 의해 연결된다. HPS는 40 MB/s의 대역폭을 갖는다.

한편, 메시지 처리기의 주소를 메시지의 헤더에 포함시킴으로써, 빠른 메시지 처리와 데이타 복사 횟수를 줄인 Active Message[13]가 T. v. Eicken과 D. E. Culler 등에 의해 연구된 바 있다. 이러한 Active Message는 Cornell 대학의 연구팀에 의해 역시 IBM SP2 위에서 구현된 바 있다[15][16]. 최근에는 IBM에 의해 Acitve Message를 지원하는 LAPI(Low-level Applications Programming Interface)가 개발되어 SP2 상에서 사용되고 있다[17].

본 실험에 사용한 LAPI는 호응하는 메시지 수신이 없어도 데이타를 보낼 수 있고, 메시지 송신 및 수신 단계에서 계산 작업과의 중첩이 가능하기 때문에, 제안하는 통신과 계산의 중첩 모델을 실험하기 적합하다. 메시지와 관련된 LAPI 함수는 LAPI_Amsend, LAPI_Put, LAPI_Get의 3가지가 있다.

6.2 실험 결과 및 고찰

이 절에서는 실험 결과에 대해 논의한다. 먼저, 각 프로그램에 대한 실행 시간 개선 정도에 대해 논의하고, 그 다음 통신/계산 중첩 시간에 대해 각각 살펴보기로 한다.

실험에 사용한 프로그램은 Lawrence Livermore Loops[18]의 kernel 1, 7과 Jacobi이다. 각 프로그램은 HPF 또는 Fortran 원시 프로그램으로부터 수작업을 통해 C 프로그램으로 변환되어 본 실험에 사용되었다

표 1 실행 시간 비교

노드수		1	2	4	6	8	16	24	32
LLL1	비중첩	0.01750	0.01483	0.01139	0.00959	0.00840	0.00828	0.00720	0.00674
	중첩	0.01760	0.01356	0.00912	0.00749	0.00664	0.00506	0.00466	0.00445
	성능 향상	-0.6%	9.4%	24.8%	28.0%	26.5%	63.7%	54.6%	51.4%
LLL7	비중첩	0.02276	0.02086	0.01424	0.01160	0.01057	0.00811	0.00811	0.00782
	중첩	0.02244	0.01953	0.01219	0.00967	0.00861	0.00615	0.00533	0.00488
	성능 향상	1.4%	6.8%	16.8%	20.1%	22.8%	31.8%	52.2%	60.1%
Jacobi	비중첩	0.27779	0.19881	0.10752	0.07443	0.06018	0.03789	0.02903	0.02660
	중첩	0.27352	0.19159	0.10051	0.06924	0.05467	0.03231	0.02379	0.02082
	성능 향상	1.6%	3.8%	7.0%	7.5%	10.1%	17.3%	22.1%	27.8%

표 1은 각 프로그램을 IBM SP 상에서 실행시킨 후 소요 시간을 비교한 것이다. 각 항목은 5회 반복 실험한 결과에서 이산점에 있는 1개를 제외하고 평균한 결과이다. 메시지 전달은 LAPI 함수를 사용하였다. 표 1에서 비중첩은 루프 세부 분할을 하지 않는 기존의 방법을 의미하고, 중첩은 본 논문에서 제안한 통신/계산 중첩 모델에 따라 프로그램을 변환한 것을 의미한다. 비중첩과 중첩의 효과를 비교하기 위해 다른 최적화 기법은 적용하지 않았다.

문제 크기는 LLL1(Lawrence Livermore Loop 1)과 LLL7에 대해 4096을, Jacobi 프로그램에 대해서는 256×256을 사용하였다. 순차 루프 반복 횟수는 20회로 하였다.

전체적으로 LLL1과 LLL7의 성능 향상이 Jacobi보다 높게 나왔는데, 이는 LLL1이나 LLL7의 상대적인 통신 비중이 Jacobi보다 크기 때문인 것으로 분석된다. Jacobi의 문제 크기는 256×256으로 LLL1이나 LLL7의 4096에 비해 16배가 크지만, 통신 소요 시간은 크게 차이가 나지 않을 것으로 생각된다. 메시지의 길이는 Jacobi, LLL1, LLL7이 각각 256, 11, 6이지만, 메시지 길이의 증가에 따르는 통신 시간의 증가는 일반적으로 매우 작다[10].

표 2는 비중첩 모델에서 각 프로그램의 계산 및 통신 소요 시간을 보여준다. 계산은 루프 자체의 실행 시간을 의미하며, 통신은 LAPI_Amsend에 의한 메시지의 송신과 수신에 걸리는 시간을 의미한다. 기타는 이외에 소요되는 시간으로 주로 수신 버퍼의 반환에 필요한 동기화에 소요되는 시간이다. Active Message와 같이 사용자 주소 공간의 버퍼로 바로 메시지 데이타를 기록할 수 있는 모델에서는 사용자 프로그램이 수신된 데이타의 사용을 마칠 때까지 동일한 수신 버퍼로 재차 메시지 수신이 이루어져서는 안된다. 따라서 송신 측에 수신 버퍼의 재사용 가능을 알려주는 동기화가 필요하다.

표 2에 의하면, 계산에 소요된 시간은 노드 수가 증가함에 따라 줄어들지만, 통신 시간은 일반적인 경향을 말할 수 없다. 실험에 사용된 프로그램들은 모두 노드 수에 관계없이 일정한 통신량을 가짐에도 불구하고 통신 시간이 불규칙하게 변하는 것은 다음 두가지 이유 때문인 것으로 추측된다. 먼저, 상호 연결망에서 다른 작업과의 충돌에 의한 지연이다. 실험을 진행할 때 노드와 통신 어댑터는 배타적으로 사용되도록 할당되지만, 스위치는 다른 노드와 공유된다. 따라서 다른 작업들의 통신 부하에 의해 영향을 받을 수 있다. 두번째 이유는 매 프로그램 실행에서 할당되는 노드가 바뀔 수 있다는 점이다. 할당되는 노드 위치에 따라 거쳐야 하는 스위치 단계 수에 차이가 있을 수 있다.

전체적으로 계산 시간과 비교했을 때, Jacobi에 비해 LLL1과 LLL7의 통신 시간 비중이 더 크다. 이것은 Jacobi는 2차원 문제인 반면, LLL1과 LLL7은 1차원 문제이기 때문이다. 그밖에 동기화 오버헤드를 포함한 기타 시간이 차지하는 비중이 크다는 것도 주목할 만하다.

통신/계산 중첩 모델에서 계산 시간과 통신 시간을 각각 측정하는 것은 매우 어려운 일이다. 통신과 계산은 중첩되어 있고, 여기서 중첩 부분을 구분하여 측정하여야 하기 때문이다. 더욱이 LAPI를 사용할 때, 통신 작업과 계산은 여러 개의 POSIX 스레드에 의해 동시적(concurrent)으로 스케쥴링되므로, 계산 작업의 경과 시간(elapased time)을 측정하더라도 여기에는 통신 작업을 위한 소프트웨어 지연 시간이 포함되어 있을 수 있다

표 3에서 비중첩과 중첩의 계산+통신 경과 시간 차이는 중첩 모델에 의한 통신 시간 감소를 어느 정도 설

표 2 비중첩 모델에서 계산 및 통신 소요 시간

노드수		2	4	6	8	16	24	32
LLL1	계산	0.009386	0.006002	0.004349	0.004765	0.002173	0.001711	0.000891
	통신	0.004512	0.003912	0.003657	0.002547	0.004028	0.003173	0.002915
	기타	0.000930	0.001474	0.001583	0.001090	0.002082	0.002317	0.002934
LLL7	계산	0.015289	0.008959	0.006694	0.005203	0.002880	0.002880	0.001710
	통신	0.004629	0.003714	0.003284	0.003663	0.002816	0.002816	0.003815
	기타	0.000945	0.001568	0.001625	0.001701	0.002410	0.002410	0.002290
Jacobi	계산	0.188656	0.096205	0.064005	0.050052	0.027758	0.018065	0.015666
	통신	0.009134	0.010160	0.009533	0.009166	0.008889	0.010169	0.009664
	기타	0.001016	0.001158	0.000893	0.000962	0.001247	0.000800	0.001268

표 3 계산+통신 경과 시간

노드수		2	4	6	8	16	24	32
LLL1	비중첩	0.013898	0.009914	0.008006	0.007312	0.006201	0.004884	0.003806
	중첩	0.013365	0.009041	0.007436	0.006162	0.002094	0.004477	0.002915
LLL7	비중첩	0.019918	0.012673	0.009978	0.008866	0.005696	0.005696	0.005525
	중첩	0.019468	0.011990	0.009603	0.008054	0.005317	0.004501	0.001517
Jacobi	비중첩	0.197790	0.106365	0.073538	0.059218	0.036647	0.028234	0.025330
	중첩	0.191327	0.100265	0.069061	0.054403	0.031370	0.022579	0.020659

명해준다. 그러나, 이 감소 폭이 전체 실행 시간 감소를 설명해주지는 않는다. 통신/계산 중첩에 의한 또다른 효과는 수신 버퍼 재사용에 따른 동기화에 소요되는 기타 시간의 감소이다. 비중첩 모델에서는 전체 루프의 실행이 끝난 후 수신 버퍼를 재사용할 수 있지만, 중첩 모델에서는 루프가 세부 분할되어 있으므로 해당 수신 버퍼를 사용하는 비지역 루프만 실행이 끝나면 바로 수신 버퍼를 재사용할 수 있다. 재사용 가능 시점이 앞당겨지기 때문에 중첩 모델에서는 동기화에 소요되는 시간이 줄어든다.

그림 8은 비중첩 모델의 실행 시간을 1로 보았을 때, 중첩 모델의 실행 시간 감소 정도와 함께 감소 요인별 비율을 보여준다. 이 그래프는 각 노드 개수에서의 수치를 평균하여 구한 것이다.

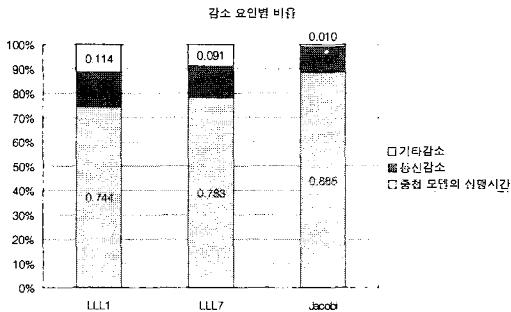


그림 8 감소 요인별 비율

종합해보면, 크게 다음 두가지 현상에 주목할 수 있다. 첫째, 전체 실행 시간 감소는 통신 시간 감소에 의해서만 이루어진 것은 아니다. 둘째, 중첩에 의해 유효 통신 시간은 감소했지만, 여전히 중첩되지 않고 남아 있는 부분이 상당히 있다. 특히 Jacobi의 경우, 전체 통신 시간을 중첩할 수 있는 충분한 계산량을 가지고 있음에도 불구하고 중첩되는 시간은 일정량을 넘지 않았다. 하나의 통신에 소요되는 시간 중 데이타 프로세서의 계산 작업과 중첩될 수 있는 시간은 한정되어 있기 때문이다.

7. 결 론

이 논문에서는 통신 시간을 줄이기 위한 방법으로 통신과 계산의 중첩 실행 모델을 제안하고 구체적인 방법을 고안하였다. 이 방법은 벡터 메시지 파이프라이닝과는 달리 함수 병렬성을 이용하지 않으며, 반복 재순서화에서 다루지 않은 구체적인 루프 반복 집합의 세부 분할 방법을 제시하였다. 또한 반복 집합의 각 세부 분할에 대한 동적인 스케쥴링 방법과 코드 생성에 대해서도 논의하였다.

병렬 루프를 기반으로 한 프로그램들에 대한 성능 평가 결과에 의하면, 본 논문의 통신과 계산 중첩 모델에 의해 전체 실행 시간은 사용된 노드 수와 프로그램에 따라 3.8 ~ 63.7 % 감소하였다. 실행 시간 감소는 통신 중첩에 의한 효과와 동기화 시간의 단축에 의해 이루어짐이 분석되었다. 동기화 시간의 단축은 통신과 계산 중첩에 의한 부수적 효과로 수신 버퍼가 조기에 재사용 가능하다는데 기인한다. 기존 방식의 실행 시간을 1로 보았을 때, 통신 중첩이 가능했던 시간의 비율은 약 0.106 ~ 0.142 정도이고, 동기화 시간의 단축은 약 0.091 ~ 0.114인 것으로 분석되었다.

수신 버퍼의 재사용 동기화는 크게 두가지 문제와 연관되어 있다. 첫째는 사용자 주소 공간으로의 직접 수신이고, 둘째는 명령형(imperative) 프로그래밍 언어들이 하나의 변수에 여러 번 쓰기를 허용하는 것이다. 이에 대한 해결로 작은 길이의 메시지에 대해서는 사용자 주소 공간으로의 직접 수신을 포기하고, 시스템 버퍼에 저장하는 방법을 취할 수 있다. 또한 각 수신되는 데이타

에 대해 새로운 사용자 주소를 할당할 수도 있다. 이와 관련한 보다 깊이 있는 연구는 향후 과제로 남는다.

본 논문에서는 이론 및 방법 면에서 통신과 계산 중첩을 제시하였다. 실험 결과를 살펴볼 때, 통신 시간 중첩에 의해 실행 시간이 상당히 감소하였음에도 불구하고 여전히 중첩될 수 없는 통신 시간이 많이 남아 있음을 알 수 있다. 이것은 하드웨어 제약에 의한 것으로, 통신시에 불가피하게 프로세서 사이클을 소비해야 하는 것이 상당 부분 존재하기 때문이다.

본 논문에서 고안한 통신과 계산 중첩 모델의 효과를 극대화하기 위해서는 이러한 하드웨어적인 재고찰이 필요하다. 통신시에 데이타 프로세서의 사이클 소비를 최소화하기 위해서는 사용자 주소 공간에 접근할 수 있는 또 하나의 통신용 프로세서를 사용하는 것도 방법이 될 수 있다. 이러한 컴퓨터 구조에 대해서는 앞으로의 연구 과제로 남는다. 향후에 이러한 컴퓨터 구조를 접목할 경우, 데이타 프로세서는 단지 통신의 개시만을 통신 프로세서에게 지시하게 될 것이기 때문에, 통신과 계산 중첩 모델의 효과를 극대화할 수 있을 것으로 기대된다.

참 고 문 헌

[1] David E. Culler, Andrea Arpaci-Dusseau et al., Parallel Computing on the Berkeley NOW, In Proc. of 9th Joint Symposium on Parallel Processing, Kobe, Japan, 1997.

[1] S. Hiranandani, K. Kennedy and C. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines," Communications of the ACM, Vol. 35, No. 8, pp. 66-80, Aug 1992.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka, Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers, Design, Implementation, and Performance Results, In Proc. of the 7th ACM Intl Conference on Supercomputing, pp. 351-360, July 1993.

[3] Zima et al., Compiling for Distributed-Memory Systems, Invited paper, In Proc. of the IEEE Special Section on Languages and Compilers for Parallel Machines, pp.264-287, Feb. 1993.

[4] High Performance Fortran Forum, High Performance Fortran Language Specification Version 2.0, The Center for Research on Parallel Computation, Jan. 1997.

[5] S. Hiranandani, K. Kennedy and C. Tseng, "Compiler Optimization for Fortran D on MIMD Distributed-Memory Machines," In Proc. of Supercomputing '91, Nov. 1991.

[6] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, "An Interactive Environment for Data Partitioning and Distribution," In Proc. of the 5th Distribution Memory Computing Conference, April 1990.

[7] M. Gerndt, "Updating Distributed Variables in Local Computations," Concurrency: Practice & Experience, Vol. 2, No. 3, pp. 171-193, Sept. 1990.

[8] S. Bokhari, "Complete Exchange on the iPSC-860," ICASE Report 91-4, Institute for Computer Application in Science and Engineering, Jan 1991.

[9] J. Li and M. Chen, "Compiling Communication-efficient Programs for Massively Parallel Machines," IEEE Trans. on Parallel and Distributed Systems, Vol. 2, No. 3, pp. 361-376, July 1991.

[10] Zhiwei Xu and Kai Hwang, "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2," IEEE Parallel and Distributed Technology, Vol. 4, No. 1, pp. 9-23, Spring 1996.

[11] A. Rogers and K. Pingali, "Process Decomposition through Locality of Reference," In Proc. of the SIGPLAN '89 Conf. on Programming Language Design and Implementation, June 1989.

[12] C. Koelbel and P. Mehrota, "Programming Data Parallel Algorithms on Distributed Memory Machines Using Kali," In Proc. of the 1991 ACM Int'l Conf. on Supercomputing, June 1991.

[13] T. von Eicken, D. E. Culler, S. C. Goldstein and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," In Proc. of the 19th Int'l Symposium on Computer Architecture, Gold Coast, Australia, May 1992.

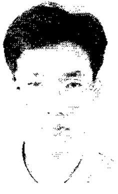
[14] C. B. Stunkel et al., "The SP2 High-Performance Switch," IBM Systems Journal, Vol. 34, No. 2, 1995.

[15] Chi-Chao Chang, Grzegorz Czajkowski and Thorsten von Eicken, "Design and Performance of Active Messages on the IBM SP2," Cornell CS Tech. Report 96-1572, Feb. 1996.

[16] Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel and Thorsten von Eicken, "Low-Latency Communication on the IBM RISC System/6000 SP," In Proc. of ACM/IEEE Supercomputing, Pittsburgh, PA, Nov. 1996.

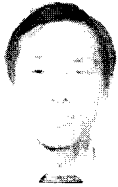
[17] Gautam Shah et al., "Performance and Experience with LAPI - a New High-Performance Communication Library for the IBM RS/6000 SP," In Proc. of Int'l Parallel Processing Symposium, 1998.

[18] F. H. McMahon, "The Livermore Fortran Kernels: a Computer Test of the Numerical Performance Range," Lawrence Livermore National Laboratory, UCRL-53745, UC Livermore, 1986.



김 정 환

1991년 2월 서울대학교 계산통계학과 이학사. 1993년 2월 서울대학교 대학원 계산통계학과 이학석사. 1999년 2월 서울대학교 계산통계학과 이학박사. 현재 삼성전자 연구원. 관심분야는 병렬처리, 컴퓨터 구조, 컴파일러.



한 상 영

1972년 서울대학교 공과대학 응용수학과 졸업(공학사). 1977년 서울대학교 대학원 계산통계학과(이학석사). 1977년 3월 ~ 1978년 3월 울산대학교 공과대학 전임강사. 1984년 3월 ~ 현재 서울대학교 자연과학대학 전산과학과 교수. 관심분야는 병렬처리임.

조 승 호

1985년 서울대학교 전자계산기공학과 졸업(학사). 1989년 서울대학교 전산과학과 대학원 졸업(석사). 1993년 서울대학교 전산과학과 대학원 졸업(박사). 1997 ~ 1999년 미국 Maryland 대학교 컴퓨터과학과 연구교수. 1993년 3월 ~ 현재 강남대학교 산업컴퓨터전자공학부 조교수. 1999년 9월 ~ 현재 (주)웹엔지니어링 대표이사. 관심분야는 데이타병렬라이브러리, 대용량 데이타 처리, 인터넷 응용 기술 등

김 흥 환

1985년 서울대학교 계산통계학과 이학사. 1987년 서울대학교 계산통계학과 이학석사. 1990년 서울대학교 계산통계학과 이학박사. 1990년 ~ 1998년 서원대학교 전자계산과 교수. 1998년 ~ 현재 건국대학교 전산과학과 교수. 관심분야는 병렬 컴퓨터 구조, 계산모델, 프로그래밍 언어임.