MPMD 방식의 동기/비동기 병렬 혼합 멱승법에 의한 거대 고유치 문제의 해법

박 필 성†

요 약

대부분의 병렬 알고리즘은 동기 알고리즘으로, 올바른 계산을 위해 작업을 일찍 끝낸 빠른 프로세서들은 동기점에서 느린 프로세서를 기다려야 하는데, 프로세서들의 성능이 다를 경우 연산 속도는 가장 느린 프로세서에 의해 결정된다. 본 논문에서는 거대 고유치 문제의 주요 고유쌍을 구하는 문제에 있어서 빠른 프로세서의 유휴 시간을 줄여 수렴 속도를 가속할 수 있는 동기/비동기 혼합 알고리즘을 고안하고 이를 MPMD 프로그래밍 방식을 사용하여 구현하였다.

A Synchronous/Asynchronous Hybrid Parallel Power Iteration for Large Eigenvalue Problems by the MPMD Methodology

Pil Seong Park†

ABSTRACT

Most of today's parallel numerical schemes use synchronous algorithms, where some processors that have finished their tasks earlier than others must wait at synchronization points for correct computation. Hence overall performance of the system is dependent upon the speed of the slowest processor. In this paper, we devise a synchronous/asynchronous hybrid algorithm to accelerate convergence of the solution for finding the dominant eigenpair of a large matrix, by reducing the idle times of faster processors using MPMD programming methodology.

키워드 : 비동기 병렬 알고리즘(Asynchronous Parallel Algorithm), 분산 메모리 시스템(Distributed Memory System), 거대 고유치 문제(Large Eigenvalue Problem), MPMD(Multiple Programs Multiple Data), MPI(Message Passing Interface)

1. 서 론

오늘날 단일 슈퍼컴퓨터로는 처리가 불가능한 거대한 문제들의 해법이 시도되고 있는데, 이들은 지리적으로 분산된 GRID 환경에서 효과적으로 실행시킬 수 있다[7]. GRID는 1990년대 중반 과학 및 공학용 분산 컴퓨팅의 연구 과정에서 등장하였으며 점차 응용분야가 넓어지고 있다. 그러나 GRID 같은 분산 환경은 기존의 단일 병렬 시스템과는 많은 점에서 다르며 이전의 기술들을 그대로 적용하기에는 무리가 있다.

기존 병렬 시스템에서는 주로 동기 알고리즘(synchronous algorithm)이 사용되는데, 직렬 연산과 같은 결과를 얻기 위해서는 동기화(synchronization)가 필요하며, 부하 균형이 필수적이다[6]. 그러나 이질 클러스터(heterogeneous cluster)처럼 프로세서들의 성능이 서로 다르거나, 지리적으로 분산된 GRID 환경에서는 이기종의 문제뿐 아니라 네트워크를 통한 메시지의 전송 지연 등으로 유휴시간이 길어질 수밖에 없다.

비동기 반복(asynchronous iteration) 알고리즘은 부하 불균형이나 네트워크를 통한 전송 지연에 의한 성능 저하를 완화하는 하나의 방법이다. 동기점을 제거한 이상적인 비동기 병렬 알고리즘을 사용하면, 각 프로세서는 유휴시간이 거의 없이 작업을 수행하며 다른 프로세서로부터 데이터가 오지 않으면 이전의 데이터를 사용하여 계산을 계속해 나간다. 따라서 동기점에서 항상 최신의 데이터를 서로 교환하고 계산을 진행하는 동기 알고리즘에 비해 전체적으로는 더 많은 CPU 시간을 요구하나 동기화를 위한 유휴시간이 거의 없으므로 프로그램의 수행에 걸리는 실제 시간(wall clock time)은 적게 걸린다[3].

현재까지 수치연산 분야의 비동기 반복 알고리즘은 선형 시스템의 해법에 국한되어 있으며, 본 논문에서 다루는 고

※ 본 논문은 해양수산부 연구개발과제로서 한국해양연구원이 수행중인 해양예보시스템 구축사업의 지원으로 수행되었다.
† 정 회 원 : 수원대학교 컴퓨터학과 교수
논문접수 : 2003년 8월 23일, 심사완료 : 2004년 2월 6일

유치 문제에 대해서는 보고된 바가 없다. 또한 클러스터나 GRID와 같은 분산 메모리 환경에서는 명시적인 메시지 교환을 통해 데이터의 교환이 이루어지는데, 현재의 모든 프로토콜들은 올바른 전송을 위해 당연히 송신과 수신이 1 : 1로 매칭되는 것을 요구하므로 있는 그대로는 비대칭적 송수신의 구현이 불가능하다. 박과 신[1]은 분산 메모리 환경에서 MPI를 사용하여 비대칭적 데이터 전송을 구현하고 이를 선형 시스템의 비동기 알고리즘에 적용하였는데, 비대칭적 송수신을 위한 기능은 프로그램 내에 흩어져 있어 복잡하고 이해와 수정이 쉽지 않다. 본 논문에서는 고유치 문제의 해법인 멱승법(power method)에 비동기적 방법을 도입한 동기/비동기 혼합 알고리즘을 고안하고 LAM/MPI 환경에서 MPMD(multiple programs multiple data) 프로그래밍 방식을 사용하여 구현하고자 한다.

2. 비동기 병렬 알고리즘 및 MPI

2.1 비동기 병렬 알고리즘

비동기 병렬 알고리즘은 여러 분야에서 개발되고 있으나 [3, 5, 12], 수치연산 분야에서는 선형 시스템의 해법에 국한되어 있으며, 고유치 문제에 대해서는 연구된 것이 없다. 이 절에서는 선형 시스템의 해법에 적용되는 비동기 반복 알고리즘에 대해 간략히 소개한다.

선형 시스템 $A\overline{x}=\overline{b}$(A는 $n\times n$, $\overline{b}$는 길이 n인 벡터)를 L개의 부분 문제로 나누는 경우, 다음과 같이 블록 형태로 쓸 수 있다. 대각 블록(diagonal block) A_{ll}은 정방행렬이며 ($l=1,\cdots,L$) 벡터 $\overline{x}$와 $\overline{b}$는 A의 소블록들의 크기에 맞도록 파티션 되었다고 가정한다.

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1L} \\ A_{21} & A_{22} & \cdots & A_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ A_{L1} & A_{L2} & \cdots & A_{LL} \end{bmatrix} \begin{bmatrix} \overline{x_1} \\ \overline{x_2} \\ \vdots \\ \overline{x_L} \end{bmatrix} = \begin{bmatrix} \overline{b_1} \\ \overline{b_2} \\ \vdots \\ \overline{b_L} \end{bmatrix} .$$

그러면 원래의 큰 문제는 L개의 작은 부분 문제들의 조합으로 나타낼 수 있다.

$$A_{ii}\overline{x_i}=\overline{b_i}-\sum_{j\neq i}^{L}A_{ij}\overline{x_j},\ \ i=1,2,\cdots,L.$$

$\overline{x_i}$는 우변의 $\overline{x_j}$($j\neq i$)와 연관되어 있으므로 한번만에 답을 구할 수 없고 오차를 점차 줄이며 반복 연산하여야 한다. 이 L개의 부분 문제들은 p개(일반적으로 $p\ll L$)의 프로세서를 가진 병렬컴퓨터의 각 프로세서에 적절히 분배하고 계산 도중 필요한 데이터를 적절히 공유 또는 교환케 하여 반복 연산함으로써 원래 문제의 해를 구할 수 있다.

공유 메모리 컴퓨터에서 프로세서 팜(processor farm) 모델을 사용할 경우 병렬 비동기 반복 알고리즘의 일반형은 다음과 같다 [11].

Algorithm 1

답이 만족스러울 때까지 다음을 반복한다.

1. i (부분 문제의 인덱스)를 선택한다.
2. $\overline{x_j}$($j\neq i$)를 공유 메모리로부터 읽는다.
3. $A_{ii}\overline{x_i}=\overline{b_i}-\sum_{j\neq i}A_{ij}\overline{x_j}$를 풀어 $\overline{x_i}$를 계산한다.
4. $\overline{x_i}$를 공유 메모리에 저장한다.

각 프로세서는 자신이 사용하는 $\overline{x_j}$의 값은, 다른 프로세서에 의한 갱신 여부에 무관하게 있는 그대로 사용한다. 따라서 빠른 프로세서는 느린 프로세서가 값을 갱신하기를 기다릴 필요가 없으며, 느린 프로세서는 빠른 프로세서가 여러 번 갱신하더라도 가장 최근의 값만을 사용하게 된다. 따라서 항상 갱신된 데이터를 동기점에서 교환한 후 다음 단계로 진행하는 동기 알고리즘에 비해, 미처 갱신되지 않은 데이터를 사용하는 경우가 많으므로 연산량 대비의 수렴 속도는 느리다. 그러나 각 프로세서는 거의 유휴 시간이 없이 연산을 수행하므로 실제 수행시간은 동기 알고리즘보다 적게 걸리며, 때로는 50%까지 빠른 결과도 보고되고 있다[3, 8].

2.2 MPI(Message Passing Interface)

메시지 패싱(message passing)은 프로세서간에 교환할 데이터를 메시지 형태로 주고받는 병렬 연산의 한 모델로서, 직렬 프로그램을 병렬화한 것에 메시지 교환을 처리할 메시지 전달 함수를 호출하는 방식으로 구현된다. MPI는 이런 함수들의 집합인 메시지 패싱 라이브러리(message passing library)의 표준으로서, 이 표준에 맞게 만들어진 MPI 라이브러리로는 각 벤더들의 상용 라이브러리 외에 LAM, MPICH, WMPI 등 여러 종류가 있다(http://www.lam-mpi.org/mpi/implementations/fulllist.php).

본 논문에서 사용하는 LAM/MPI 환경에서는 계산 주체가 CPU가 아니라 프로세스이며 사용자는 프로그램 실행시 필요한 프로세스의 개수를 지정하게 된다. 각 프로세스는 랭크(rank)에 의해 구분되며, n개의 프로세스를 사용할 경우 0부터 $n-1$의 값을 가진다.

MPI의 대표적인 메시지 송신 및 수신 함수는 다음과 같다.

```
int MPI_Send(void * buffer, int count, MPI_Datatype datatype,
             intdestination, int tag, MPI_Comm communicator)
int MPI_Recv(void * buffer, int count, MPI_Datatype datatype, int
             source, int tag,
             MPI_Comm communicator, MPI_Status * status)
```

buffer는 송신/수신될 데이터의 저장 장소, datatype은 전송 데이터의 형, count는 전송 데이터의 수, destination과 source는 각기 수신자 및 발신자의 랭크, tag은 메시지를

구분하는 번호, communicator는 송수신에 참여하는 프로세스들의 집합, 그리고 status는 수신 결과와 관련된 정보를 가지는 구조체이다. 송신자에 의한 MPI_Send() 함수의 호출은 수신자의 MPI_Recv() 호출에 의해 처리되어야 하며, 이 경우 count, datatype, tag, communicator가 서로 일치해야 하고, destination과 source는 서로 상대방의 랭크를 사용한다.

3. 다루는 문제 및 비동기화를 위한 착상

멱승법(power method)은 비록 수렴속도는 느리나 행렬의 주요 고유벡터를 구하는 알고리즘 중 가장 간단하고 안정적이며 이론적으로 가장 잘 이해된 것이다. 본 논문에서는 [9]에 기술된 2큐 오버플로 대기망 모델(overflow queuing network model) 문제의 행렬 Q를 다룬다. 행렬의 크기는 10,000×10,000이나 5대각 희소 행렬(5-diagonal sparse matrix)로서, $Q\bar{x}=\lambda\bar{x}$의 주요 고유벡터를 구하는 것이 목적이다. 그러나 행렬 Q는 있는 그대로는 멱승법이 수렴하지 않을 수도 있으므로 Q의 대각성분에 작은 값 1을 더함으로써 새로운 문제 $A\bar{x}\equiv(Q+I)\bar{x}=(\lambda+1)\bar{x}$는 유일한 주요 고유치 $\mu=\lambda+1$을 가지며 주요 고유벡터는 Q의 것과 동일하다.

고유치 문제 해법의 수렴 여부를 판단하는 방법은 여러 가지 있을 수 있으나 본 논문에서는 다음의 잔차(residual) 벡터를 정의하고 그 크기를 수렴의 척도로 사용하기로 한다.

[정의 1] 고유치 문제 $A\bar{x}=\lambda\bar{x}$에서 행렬 A의 주요 고유치와 고유벡터를 각기 λ^* 및 $\bar{x}^*$라 하자. 그러면 $\bar{x}^*$의 근사 벡터 $\bar{x}$의 잔차 벡터는 $\bar{r}=A\bar{x}-\lambda\bar{x}$(단 $\|\bar{x}\|_2=1$, λ는 λ^*의 근사치)로 정의한다.

이 척도를 사용한 직렬 멱승법은 다음과 같다.

Algorithm 2 (직렬 멱승법)

1. Choose $\forall\,\bar{x}$, 단 $\|\bar{x}\|_2=1$.
2. $\bar{z}\leftarrow A\bar{x}$.
3. Repeat the following :
   1) $\bar{x}\leftarrow\bar{z}/\|\bar{z}\|_2$
   2) $\bar{z}\leftarrow A\bar{x}$
   3) $\lambda\leftarrow\bar{x}\cdot\bar{z}$
   4) Compute $\|\bar{r}\|_2$, and if it is small enough then stop.

단계 3.1)에서 벡터 $\bar{z}$를 그것의 크기로 나누는 것은, 벡터 $\bar{x}$의 모든 성분에 행렬 A가 같은 회수로 곱해지는 것을 보장하며, §4의 동기 병렬 알고리즘의 동기화에 해당된다. 따라서 여러 프로세스가 완전히 비동기적으로 연산을 수행한다면 결코 직렬 멱승법과 같은 결과를 얻을 수 없으므로 어느 정도 동기화가 필요한데, 어떻게 동기화를 최소화하고 수렴을 가속화하는가 하는 것이 문제이다.

정칙 선형 시스템은 유일한 해를 가지므로 행렬이 [11]에 주어진 조건만 만족하면 §2.1의 비동기 알고리즘의 수렴이 보장되나, 고유치 문제의 경우에는 수렴의 조건에 대해 알려진 것이 없다. 또한 어떤 고유벡터의 임의의 상수배도 역시 고유벡터가 되므로 무한히 많은 수의 해가 존재하고 각 프로세서가 비동기적으로 계산한 부분벡터들 각각은 어떤 고유벡터의 일부분으로 수렴할지 모르나 이들을 단순히 조합한 전체 벡터는 그 어떤 고유벡터와는 다른 엉뚱한 벡터가 된다.

예를 들어, 초기벡터 $\overline{x_0}$를 사용하여 행렬 A의 주요 고유벡터를 구하기 위해 직렬 멱승법 $\overline{x_{i+1}}=A\overline{x_i}$을 적용하여 주요 고유벡터 $\overline{x_\infty}$를 계산하는 다음의 경우를 보자.

$$A=\begin{bmatrix}3&-2&1\\-2&3&1\\0&1&5\end{bmatrix},\quad \overline{x_0}=\begin{bmatrix}1\\1\\1\end{bmatrix},\quad \overline{x_\infty}=\begin{bmatrix}1\\1\\4.2363\end{bmatrix}$$

<표 1>은 정규화하지 않는 멱승법(정규화는 연산 도중 오버플로나 언더플로 되는 것을 방지하는 역할만 할 뿐 벡터의 각 성분의 비율에는 영향이 없다.)을 적용하여 계산된 벡터및 이전 벡터와의 성분별 비율을 나타낸다. 이를 보면, 각 성분은 처음에는 각기 다른 비율로 변화하나 점차 모두 같은 비율 5.2361로 수렴함을 알 수 있다.

각 성분을 세 개의 프로세스가 독립적으로 계산할 경우, 셋째 프로세스가 다른 것의 1/3의 성능을 가진다고 하자. 그러면 빠른 프로세스가 3회의 멱승 연산하는 t초 동안 느린 프로세스는 단 1회만 연산하므로 t 및 $2t$초 후 각 프로세스가 비동기적으로 계산한 결과를 단순히 조합하면 (40,

〈표 1〉 정규화하지 않은 멱승법 시행의 결과

i	0	1	2	3	4	5	6	7	8
$\overline{x_i}$	1 1 1	2 2 6	8 8 32	40 40 168	208 208 880	1088 1088 4608	5696 5696 24128	29824 29824 126336	156160 156160 661504
이전 벡터와의 성분비		2 2 6	4 4 5.333	5 5 5.25	5.2 5.2 5.2381	5.2308 5.2308 5.2364	5.2353 5.2353 5.2361	5.2360 5.2360 5.2361	5.2361 5.2361 5.2361

40, 6) 및 (5696, 5696, 32)가 얻어져 성분비는 구하려는 $\overline{x_\infty}$와는 전혀 다른 벡터가 얻어진다. 따라서 동기화가 필요하며 동기화 단계 사이에 각 프로세스가 수행한 멱승 연산의 회수를 감안하여 느린 프로세스의 결과를 빠른 프로세스와 같은 횟수의 멱승 연산을 수행한 것으로 시뮬레이션 해야 한다.

이를 위해 각 프로세스는 벡터 성분들의 변화 비율과 1회의 멱승 연산에 걸린 시간을 추적하고 가장 느린 프로세스가 1회의 연산을 끝낼 때마다 한번씩 동기화를 하도록 한다. 물론 빠른 프로세스는 그 동안 기다리지 않고 독자적으로 추가적인 멱승 연산을 더 수행하도록 한다. 예를 들어, 느린 프로세스가 1회 멱승 연산하여 얻은 값을 사용하여 다른 프로세스처럼 3회 멱승 연산한 것과 유사한 값을 얻으려면 다음과 같이 시뮬레이션 할 수 있다.

(느린 프로세스가 1회 멱승 연산한 결과 값) × (계산된 성분비)^2

이렇게 하여 t초 및 $2t$초 후에 동기화하여 조립된 전체 벡터는 (40, 40, 216) 및 (5696, 5696, 32762)로서 비록 그리 정확하지는 않으나 이전보다 훨씬 나은 결과를 준다.

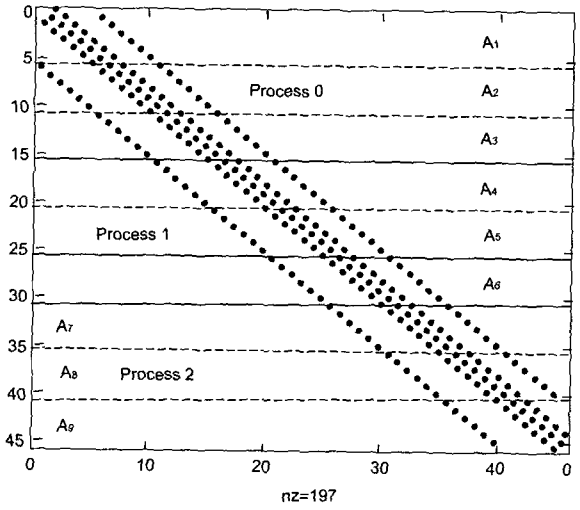
한편 이런 전략은 처음부터 적용할 것이 아니라, 가능한 한 각 성분이 어느 정도 수렴한 이후 적용하는 것이 좋다. 예를 들어, $\overline{x_5}$까지는 §4처럼 동기적 방법으로 계산하고, 그 이후는 비동기적으로 계산하며 느린 프로세스가 1회 멱승 연산한 후 동기화하여 얻어지는 벡터 $\overline{x_6^A}$은 §4의 동기 병렬 멱승법(이 경우 빠른 프로세스는 동기점에서 $2t/3$동안 기다려야 한다)에 의해 얻어진 $\overline{x_6}$보다 구하려는 $\overline{x_\infty}$의 성분비에 더 가까움을 알 수 있다.

$\overline{x_6^A}$ = (156160, 156160, 661586 ≈ 24128×5.2361^2)
⇒ 성분비 1 : 1 : 4.23611
$\overline{x_6}$ = (5696, 5696, 24128) ⇒ 성분비 1 : 1 : 4.23596

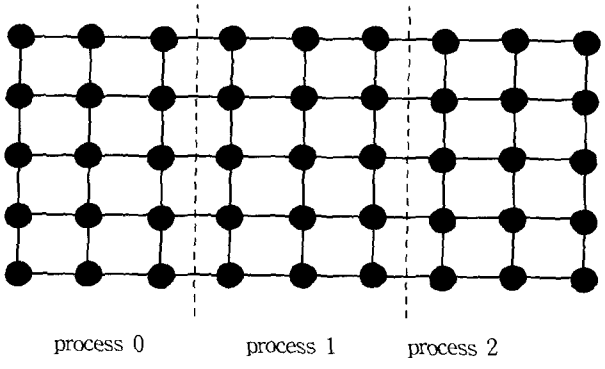
4. 동기 병렬 멱승법

비동기 병렬 알고리즘을 고안하기 위해서, 우선 동기 병렬 알고리즘을 고안하고 동기점들을 제거한다. 이를 위해 축소된 문제의 경우를 보자. 예를 들어, 첫째 및 둘째 큐의 상태 수가 각기 9 및 5인 대기망 문제의 행렬은 크기가 45 × 45인 5대각 희소 행렬로서 0이 아닌 성분들만을 점으로 표시하면 (그림 1)과 같다.

구하는 고유벡터의 각 성분들을 하나의 절점으로, 각 성분들간의 연관 관계를 간선으로 하여 그래프로 나타낸 것은 (그림 2)와 같다. 즉 그래프의 k번째 수직선상의 절점들은 위로부터 각기 고유벡터의 성분 $x_{5(k-1)+1}$부터 x_{5k}에 해당된다. 따라서 이는 5×9의 2차원 격자 문제의 행렬과 구조가 같고, 이를 3개의 프로세스가 균등하게 나누어 계산할 경우, 각 프로세스가 계산할 미지수들은 3개의 수직선상의 15개의 절점들로서 그림에서 점선으로 나누어져 있다. 좌측 및 우측의 경계에서의 값은 인접 프로세스의 우측 및 좌측 경계점의 값과 서로 영향을 미치므로 그 값들이 업데이트될 때마다 인접 프로세스와 최신 값을 서로 교환해야 한다.



(그림 1) 예제 문제의 행렬 구조와 부분 행렬로의 분할



(그림 2) 각 프로세스가 담당하여 계산할 절점들

한편 (그림 2)의 i번째 수직 격자선 상의 5개의 격자점의 연산과 관련된 것은 (그림 1)에서 가로선으로 구분된 부분행렬 A_i($i=1, \cdots, 9$)이다. 따라서 각 프로세스는 이들 중 3개(즉 15개 행)씩만 가지면 된다(즉 프로세스 0은 A_1, A_2, A_3 프로세스 1은 A_4, A_5, A_6, 프로세스 2는 A_7, A_8, A_9). 이것에 맞추어 벡터 $\overline{x}$ 및 $\overline{z}$를 파티션할 경우, 길이가 5인 부분 벡터들을 각기 $\overline{x_1}, \overline{x_2}, \cdots, \overline{x_9}$ 및 $\overline{z_1}, \overline{z_2}, \cdots, \overline{z_9}$이라 하자.

실제 우리가 다루는 문제는 10,000×10,000이므로, 부분 행렬들 A_i($i=1, \cdots, 100$)의 크기는 100×10,000, 각 부분 벡터 $\overline{x_i}$ 및 $\overline{z_i}$($i=1, \cdots, 100$)의 길이는 100이다. 멱승법의 한

단계 $\bar{z} \leftarrow A\bar{x}$에 의해 계산되는 벡터 $\bar{z}$는 다음 100개의 부분 멱승법의 조합이다.

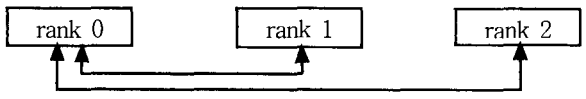
$$\bar{z_i} \leftarrow A_i\bar{x},\ i=1, 2, \cdots, 100,$$

이 100개의 부분 문제를 p개($p<100$)의 프로세스를 사용하여 계산하는 경우, k번째 프로세스는 인덱스 i의 값이 s_k부터 e_k까지의 부분 멱승법을 계산한다고 하자(예를 들어, (그림 2)에서는 프로세스 1의 경우, $s_1=4$, $e_1=6$). 이 경우,

Algorithm 3 (k번째 프로세스의 동기 병렬 알고리즘)

1. Take an initial guess for $\bar{x}$.
2. $\bar{z_i} \leftarrow A_i\bar{x}$, $i=s_k$ to e_k.
3. Collect $\bar{z_i}$ to form $\bar{z}$, and distribute it to all proc esses.
4. Repeat the following :
   1) Compute $\|\bar{z}\|_2$.
   2) $\bar{x} \leftarrow \bar{z}/\|\bar{z}\|_2$
   3) $\bar{z_i} \leftarrow A_i\bar{x}$, $i=s_k$ to e_k.
   4) Collect $\bar{z_i}$ to form $\bar{z}$, and distribute it to all processes.
   5) $\lambda \leftarrow \bar{x} \cdot \bar{z}$
   6) Check if $\|\bar{r}\|_2 = \|A\bar{x}-\lambda\bar{x}\|_2 = \|\bar{z}-\lambda\bar{x}\|_2$ is small enough.

모든 프로세스는 연산을 독립적으로 수행하되, 그 중 하나는 단계 3 및 4.4)에서 자신 및 다른 프로세스가 계산한 부분 벡터 $\bar{z_i}$를 모아 전체 벡터 $\bar{z}$를 생성하고 다른 프로세스에게 전달하는 역할도 겸해야 하는데, 편의상 이를 매스터 프로세스라 부르도록 한다. 예를 들어 (그림 2)의 경우처럼 3개의 프로세스를 사용해 계산하는 경우, rank 0을 매스터 프로세스로 지정하면, 각 프로세스 간의 데이터 교환은 (그림 3)과 같다.



(그림 3) 3개의 프로세스를 사용하는 경우의 메시지 패싱

이 경우 프로세스 간의 데이터 교환이 일정하게 일어나므로 같은 프로그램을 모든 프로세스가 시행하는 SPMD(single program multiple data) 방식으로 쉽게 구현할 수 있다.

5. 동기/비동기 혼합 병렬 멱승법

§3에서 언급했듯이, 어느 정도 수렴할 때까지는 동기 병렬 멱승법을 사용하고 그 이후 비동기 멱승법으로 전환하는 혼합 알고리즘을 사용하도록 한다. 즉 비동기 연산 부분에서는 느린 프로세스가 1회 멱승 연산할 때마다 동기화하되, 빠른 프로세스들은 그 동안 쉬지 않고 정규화하지 않는 멱승 연산을 수행하고 느린 프로세스의 연산 결과는 빠른 프로세스의 것에 맞추어 시뮬레이션한 결과를 사용토록 한다.

이를 위해서, 각 프로세스는 자신이 1회 멱승 연산하는 데 걸리는 시간과 자신이 담당하는 부분 벡터의 갱신 전후 크기의 비율을 계산하여 동기화 시점에서 보고토록 한다. 부분 벡터의 갱신 전후 크기의 비율은 다음과 같이 정의한다.

[정의 2] 벡터 $\bar{x}$의 어느 부분 벡터를 $\bar{x_p}$라 하자. 이 부분 벡터에 1회 멱승 연산한 결과를 $\bar{x_p}' = A_p\bar{x}$라 할 때, 1회 멱승 연산에 의한 크기의 비율은 $\|\bar{x_p}'\|_2 / \|\bar{x_p}\|_2$로 정의한다.

이런 비동기 요소를 도입한 혼합 알고리즘의 골격은 다음과 같다.

Algorithm 4 (동기/비동기 혼합 병렬 알고리즘)

1. 모든 프로세스는 Algorithm 3의 동기 병렬 알고리즘을 일정한 회수 시행한다. 이 때 각 프로세스는 매번 1회 멱승 연산에 걸린 시간과 크기 비율을 계산한다.
2. 각 프로세스는 Algorithm 5 또는 6을 비동기적으로 수행하되, 가장 느린 프로세스의 1회 멱승 연산에 맞추어 동기화하며 계산한다.

§4의 동기 알고리즘에서는 모든 메시지 패싱이 동기화되므로 매스터 프로세스는 언제 어느 프로세스로부터 어떤 종류의 메시지가 도달할 지 알 수 있다. 그러나 Algorithm 4의 비동기 연산 부분에서는 그 예측이 불가능하다. 이런 문제를 [1]에서는 다음과 같은 코드를 프로그램 곳곳에서 넣어 수시로 호출하는 방식으로 해결하였다.

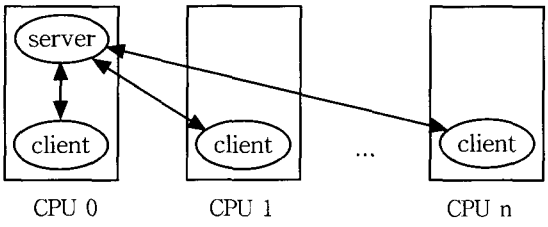
```
MPI_IProbe(MPI_ANY_SOURCE, MPI_ANY_TAG,
com municator, &flag, &status) ;
If (flag) { /* flag is TRUE if there is a message arrived. */
        메시지 종류에 따라 적절한 작업을 수행
}
```

이는 송신자 및 송신 메시지의 종류와 무관하게(MPI_ANY_SOURCE, MPI_ANY_TAG) 자신에게 온 메시지가 있는지 확인하고 있으면 수신하고 처리하도록 한다. 이를 자주 호출할수록 송신자의 대기 시간은 줄어드나 이 함수의 잦은 호출은 대부분 시간만 낭비할 뿐이다. 이처럼 비동기적으로 연산이 수행될 경우, 매스터 프로세스가 그 진행 상황을 체크하고 다른 프로세스와 데이터를 주고받으며 또한 자신의 연산을 수행하도록 하는 SPMD(single program

multiple data) 방식으로 구현하면 프로그램이 아주 복잡해진다.

이런 문제는, 프로세스의 종류에 따라 각기 다른 프로그램을 사용하는 MPMD(multiple programs multiple data) 방식을 사용하여 해결할 수 있다. MPMD 방식은 빠른 개발 싸이클, 코드의 재사용, 모듈 프로그래밍이 필요한 경우, 그리고 불규칙한 작업 로드나 데이터 전송 패턴이 필요할 때 아주 유용하며 SPMD 방식에 비해 프로그램이 간결하다[4].

편의상 실제 멱승 연산을 수행하는 프로세스를 클라이언트, 클라이언트들이 보고하는 정보를 바탕으로 전체병렬 연산의 수행을 통제하고 클라이언트가 필요로 하는 데이터를 공급하는 프로세스를 서버라 부르도록 한다. 본 논문에서는 모든 CPU에 하나씩의 클라이언트를 띄우며 서버는 첫 번째 CPU에 하나만 띄운다. 그리고 (그림 4)처럼 클라이언트끼리 직접 송수신하지 않고 첫 번째 CPU 상의 서버와 데이터를 주고받으며 시행하도록 한다.



(그림 4) MPMD 방식으로 구현할 경우의 메시지 패싱
(타원 : 프로세스, 직사각형 : CPU)

편의상 k번째 클라이언트가 업데이트하는 부분벡터들 $\overline{x_{k_1}}, \cdots, \overline{x_{k_m}}$로 구성된 부분벡터를 $\overline{x_K}$로 나타내면 클라이언트의 비동기 알고리즘은 다음과 같다.

Algorithm 5 (k번째 클라이언트의 비동기 알고리즘)

1. 다음을 반복한다.
   1) $\overline{x_{k_1}}, \cdots, \overline{x_{k_m}}$(즉 $\overline{x_K}$)을 하나씩 순서대로 서버가 지정한 회수만큼 동기화하지 않는 멱승 연산을 수행하며 걸린 시간을 측정한다.
   2) 1회 멱승 연산에 의한 $\overline{x_K}$ 크기의 변화 비율을 계산한다([정의 2] 참조).
   3) $\overline{x_K}$와 소모된 시간, $\overline{x_K}$ 크기의 변화 비율을 서버에게 보낸다.
   4) 동기화된 전체 벡터 $\bar{x}$ 및 다음 동기화까지 수행할 멱승 연산의 회수를 서버로부터 전달받는다.
   5) 서버로부터 수신한 신호가 STOP이면 계산을 끝낸다.

Algorithm 5의 시행 이전에 일정 회수의 동기 멱승법이 이미 시행되었으므로 서버는 각 클라이언트의 속도를 이미 알고 있고, 단계 1.1)에서 클라이언트도 이미 서버로부터 동기화하기 전에 몇 회의 멱승 연산을 해야 하는지 알고 있다. 한편 메시지 패싱에 소요되는 시간을 줄이기 위해, 벡터 $\bar{x}$를 저장할 배열은 실제 크기보다 2만큼 더 잡아 멱승 연산에 소모된 시간과 부분 벡터 크기의 변화 비율을 저장하여 모든 것을 한꺼번에 주고받도록 한다. 또한 서버는 계산이 충분히 수렴하면 각 클라이언트가 계산을 끝내라는 신호로서 배열의 첫째 값 x[0]에 미리 정해진 특수한 값을 담아 보내도록 한다.

Algorithm 6 (서버의 비동기 알고리즘)

답이 충분히 수렴할 때까지 다음을 반복한다.

1. 클라이언트로부터 오는 메시지를 기다린다.
   1) 송신자와 보내온 메시지의 종류를 파악하고 저장한다.
   2) 모든 클라이언트로부터 갱신된 데이터를 받으면
      a. 클라이언트의 멱승 시행 회수에 맞추어 데이터를 시뮬레이션하여(§3 참조) 전체 벡터를 구성하고 수렴 여부를 판단한다.
      b. 가장 느린 클라이언트의 속도에 맞추어 각 프로세스가 독립적으로 멱승 연산할 회수를 계산한다.
      c. 다음의 데이터를 모든 클라이언트에게 전송한다.
         Case 1. 충분히 수렴하였으면
                 STOP 신호를 x[0]에 실어 보낸다.
         Case 2. 그렇지 않으면
                 전체 벡터 $\bar{x}$와 다음 동기화까지의 멱승 연산의 회수를 보낸다.

서버 프로세스는 §4처럼 MPI_Iprobe()를 수시로 호출하는 대신, 다음과 같이 클라이언트의 메시지를 기다리다가 메시지가 도착하면 즉시 처리하므로 불필요한 함수 호출에 시간을 낭비하지 않으며 CPU 시간을 아주 적게 소모한다.

```
MPI_Recv(buffer,count,MPI_Datatype,MPI_ANY_SOURCE,MPI
  _ANY_TAG,MPI_COMM_WORLD,status)
sender = status.MPI_SOURCE ; /* 송신자를 파악 */
jobkind = status.MPI_TAG ; /* 송신 메시지의 종류를 파악 */
```

6. 성능 실험 및 고찰

Algorithm 3과 Algorithm 4의 성능를 비교하기 위해 LAM/MPI [10]를 사용하여 수원대학교의 Hydra 클러스터에서 수치 실험하였다. Hydra는 11대의 노드로 구성된 고성능 HPC 클러스터로서, 각 노드는 Intel SHG2 보드에 2.2 GHz Xeon CPU와 메모리(매스터 노드는 2GB, 슬레이브 노드는 1GB)가 장착되어 있으며, 3Com 4900 series Gigabit 스위치로 상호 연결되어 있다. 그러나 Hydra는 균질 클러스터이므로, 노드의 성능 차이에 의한 반응을 보기 위해 마지막 노드 상의 클라이언트에게 계산 결과에는 영향을 미치지 않으나 단순히 시간만 소모하는 추가적인 벡터

의 계산을 부여하여 시뮬레이션 하였다.

<표 2>는 6개의 노드를 사용하여 잔차가 10^{-9}에 도달할 때까지 걸린 실제 시간을 나타낸다(비록 클러스터는 독점 사용하였으나 다른 시스템 작업등에 의한 영향을 배제하기 위해 거의 같은 값이 나올 때까지 5회 이상 시행하여 평균한 결과이다). 첫 칸은 마지막 노드에 부여한 추가적 부하를 나타내며, 둘째 및 셋째 칸은 동기 알고리즘이 소모한 실제 시간과 빠른 프로세스의 유휴 시간 비율, 그리고 대략적으로 추정한 빠른 프로세스와 느린 프로세스의 성능의 비(f/s)를 나타낸다. 한편 넷째 칸의 Pre-sync는 Algorithm 4의 단계 1(즉 비동기 멱승법을 시행하기 전에 수행하는 동기 멱승법)의 회수를 나타낸다. 마지막 세 칸의 Sync ratio는 Algorithm 4의 2단계 비동기 연산 부분에서 동기점 사이에 빠른 프로세스와 느린 프로세스가 수행하는 멱승 연산 횟수의 비율을 나타낸다. 이는 각 클라이언트가 1회 멱승 연산 수행에 걸린 시간을 근거로 동적으로 동기점을 조정하도록 할 수 있으나, 본 실험에서는 다양한 비율에 따른 반응을 보기 위해 임의로 고정된 값을 사용하였다.

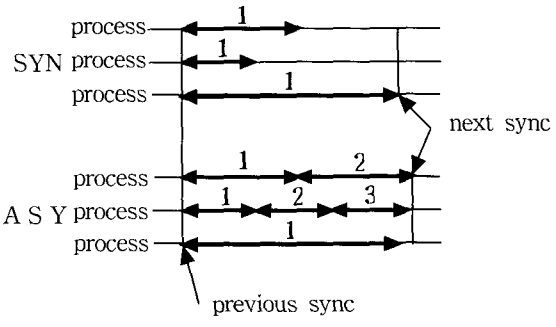
〈표 2〉 잔차의 크기가 10^{-9}에 도달할 때까지 걸린 실제 시간

Extra load	Synchronous wall clock		Hybrid wall clock time			
				Sync. ratio		
	time	idle % (f/s)	Pre-sync	2	3	4
0	93.18	0 %	-			
10	169.49	45.0% (1.82)	500	129.24	159.09	202.48
			1000	137.07	161.81	205.00
			1500	136.21	160.31	202.69
			2000	135.19	160.26	200.03
			2500	135.31	158.03	197.39
			3000	134.66	157.39	194.06
15	208.04	55.2% (2.23)	500	156.36	168.40	210.38
			1000	164.29	172.83	214.39
			1500	164.83	171.83	211.23
			2000	164.04	171.58	210.42
			2500	163.94	170.78	207.77
			3000	163.19	169.64	205.70
20	246.65	62.1% (2.65)	500	183.39	184.66	217.77
			1000	193.15	188.44	222.03
			1500	193.22	189.12	221.28
			2000	192.90	189.11	219.39
			2500	192.38	188.05	218.44
			3000	191.94	188.15	217.24
25	285.53	67.4% (3.06)	500	211.05	212.54	225.47
			1000	221.98	216.72	229.61
			1500	221.62	217.88	230.29
			2000	221.57	217.49	229.75
			2500	221.67	216.87	228.75
			3000	220.55	216.84	227.75

동기 알고리즘의 경우, 추가 부하가 없을 때 93.18초가 소요되었고, 마지막 클라이언트의 부하를 늘릴수록 걸린 시간이 거의 선형으로 증가한다. 동기/비동기 혼합 알고리즘의 경우, pre-sync의 회수에 따른 영향은 크지 않으나 500회 시행했을 경우 가장 좋은 결과를 나타내는데, 이는 이미 어느 정도 충분히 수렴했음을 의미하며, 더 많은 pre-sync 회수를 시행할 경우에는 빠른 프로세스는 동기화를 위해 유휴 시간만 낭비했음을 말해 준다.

그러나 모든 요인 중 sync ratio의 비가 가장 중요한 것으로 사료된다. 즉 빠른 프로세스의 유휴 시간이 45~55% (추가 부하 10 또는 15)인 경우 sync ratio가 2인 경우 가장 효율적이며 그 비율이 커질수록 성능이 저하된다. 이는 동기화 할 때까지 빠른 프로세스가 더 많은 멱승 연산을 수행하는 동안 느린 프로세스가 유휴 시간을 가지기 때문이다. 예를 들어 추가 부하가 10인 경우, 빠른 프로세스의 성능은 느린 프로세스의 약 1.82배이므로 빠른 프로세스가 2회 연산할 때마다 동기화(느린 프로세스도 약간의 유휴 시간을 가지게 되지만)하는 것이 가장 적절할 것이며, 3회나 4회마다 동기화한다면 느린 프로세스는 더 많은 유휴 시간을 가지게 되어 성능이 떨어진다. 한편 추가 부하가 25인 경우, 빠른 프로세스의 성능은 느린 프로세스의 약 3.06배이므로 sync ratio가 3인 경우 가장 효율적임을 볼 수 있다.

이상의 분석은 연산에 걸린 실제 시간만을 바탕으로 한 것으로 다른 요인들도 영향을 미친다. 즉 sync ratio를 크게 하면(빠른 프로세스의 느린 프로세스에 대한 성능 비의 범위 내에서) 더 자주 시뮬레이션된 값에 의존하게 되므로 계산치는 부정확하게 되는 반면, 빠른 프로세스의 유휴 시간이 활용되어 더 나은 결과를 얻을 수 있다는 상반된 결과를 준다.



(그림 5) 동기점 사이 두 알고리즘의 프로세스별 멱승 연산 회수

(그림 5)는 프로세스의 성능 비가 대체적으로 2 : 3 : 1인 경우, 동기 알고리즘(SYN)과 비동기 연산 부분의 혼합 알고리즘(ASY)이 두 동기점 사이에서 각 프로세스가 수행하는 멱승 연산의 회수 및 유휴 시간(화살표로 표시되지 않은 부분)을 나타낸다. SYN의 경우 프로세스 0과 1의 유휴

시간이 거의 1/2 및 2/3인 반면, ASY의 경우 이들은 거의 유휴 시간 없이 각기 2회 및 3회 멱승 연산을 수행하게 되므로(반면, 느린 프로세스 2는 약간의 유휴 시간을 가지게 된다) 수렴이 가속화되어 전체적으로 시간이 적게 걸릴 것임을 짐작할 수 있다.

7. 결론 및 향후 연구

본 논문에서는 장차 GRID처럼 같은 지리적으로 분산된 계산 자원을 사용하거나 이질 클러스터를 사용할 경우 문제가 될 노드들의 성능 차이 및 네트워크에 의한 전송 지연을 완화할 수 있는 하나의 방안으로 MPMD 방식의 동기/비동기 혼합 멱승 알고리즘을 고안하고 이를 10,000 × 10,000의 거대 희소행렬 문제에 적용하였다. 그 결과 새로운 동기/비동기 혼합 멱승 알고리즘은 빠른 프로세서들이 유휴 시간을 효율적으로 이용하게 함으로써 순수한 동기 멱승 알고리즘에 비해 수행 시간을 거의 25%까지 감소시켰다.

그러나 본 연구에서는 비동기 연산을 시작하기 전에 수행하는 동기 멱승 연산의 회수와 sync ratio의 영향을 보기 위해 다양한 값으로 고정하여 수행하였다. 따라서 실제 적용에 있어서는 최적의 결과를 얻기 위해 이런 인자를 동적으로 결정하여 수행하는 보다 정교한 알고리즘의 개발이 필요하다.

참 고 문 헌

[1] 박필성, 신순철, "비동기 알고리즘을 이용한 분산 메모리 시스템에서의 초대형 선형 시스템 해법의 성능 향상", 정보처리학회논문지A, 제8A, pp.439-446, 2001.

[2] B. Baràn, E. Kaszkurewicz and A. Bhaya, "Parallel asynchronous team algorithms : Convergence and performance analysis," IEEE Transactions on Parallel & Distributed Systems, Vol.7, pp.677-688, 1996.

[3] R. Bru, V. Migallón, J. Penadés and D. B. Szyld, "Parallel, synchronous and asynchronous two-stage multisplitting methods," Electronic Transactions on Numerical Analysis, Vol.3, pp.24-38, 1995.

[4] C. Chang, G. Czajkowski, T. von Eicken and C. Kesselman, "Evaluating the performance limitation of MPMD communication," In Proceedings of SC '97, San Jose, CA, pp.15-91, November, 1997.

[5] R. Cole, and Z. Ofer, "An asynchronous parallel algorithm for undirected graph connectivity," TR-546, Dept., of Computer Science, New York University, Feb., 1991.

[6] I. T. Foster, 'Designing and building parallel programs,' Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[7] I. Foster, C. Kesselman and S. Tuecke, "The anatomy of the Grid : Enabling scalable virtual organizations," J. Supercomputer Applications, 15(3), 2001.

[8] A. Frommer, H. Schwandt and D. B. Szyld, "Asynchronous weighted additive Schwarz methods," Electronic Transactions on Numerical Analysis, Vol.5, pp.48-67, 1997.

[9] L. Kaufman, "Matrix methods for queuing problems," SIAM Journal on Scientific & Statistical Computing, Vol.4, pp.525-552, 1983.

[10] Ohio Supercomputing Center, 'MPI Primer/Developing with LAM,' 1996.

[11] D. B. Szyld, "Different models of parallel asynchronous iterations with overlapping blocks," Computational and Applied Mathematics, Vol.17, pp.101-115, 1998.

[12] A. Uresin and M. Dubois, "Parallel asynchronous algorithms for discrete data," Journal of ACM, Vol.37, pp.588-606, 1990.

박 필 성

e-mail : pspark@suwon.ac.kr

1977년 서울대학교 해양학과(학사)

1978년~1982년 KIST 부설 해양연구소 연구원

1984년 미국 올드 도미니언 주립대학 계산학 및 응용수학과 석사

1991년 미국 메릴랜드 주립대학 응용수학과 박사

1991년~1995년 한국해양연구소 선임연구원(전산실장, 수치모델연구그룹장)

1995년~현재 수원대학교 컴퓨터학과 부교수

관심분야 : 수치 선형대수, 고성능 컴퓨팅, 클러스터링, 큐잉 문제의 수치해법 등