# 비동기 알고리즘을 이용한 분산 메모리 시스템에서의 초대형 선형 시스템 해법의 성능 향상

박 필 성† · 신 순 철††


## 요 약

현재 대부분의 병렬 알고리즘은 동기 알고리즘으로 올바른 계산을 위해서는 프로세서들의 동기화와 부하균형이 필수적이다. 만일 부하균형이 불가능하거나 이질적 클러스터처럼 각 프로세서의 성능이 다른 경우, 연산은 가장 느린 프로세서의 성능에 의해 결정된다. 비동기 반복법은 이런 문제를 해결하는 하나의 방안으로 각광받고 있으나, 현재까지의 연구는 비교적 구현이 쉬운 공유 메모리 시스템을 사용한 것이었다. 본 논문에서는 분산 메모리 환경에서 초대형 선형 시스템 문제를 풀기 위해, 빠른 프로세서의 유휴 시간을 최대한 줄임으로써 전체적으로 성능을 향상시키는 비동기 병렬 알고리즘을 제안하고 이를 클러스터에 구현하였다.


# Improving Performance of Large Sparse Linear System Solvers On Distributed Memory Systems By Asynchronous Algorithms

Pil Seong Park† · Soon Churl Shin††


## ABSTRACT

The main stream of parallel programming today is using synchronous algorithms, where processor synchronization for correct computation and workload balance are essential. Overall performance of the whole system is dependent upon the performance of the slowest processor, if workload is not well-balanced or heterogeneous clusters are used. Asynchronous iteration is a way to mitigate such problems, but most of the works done so far are for shared memory systems. In this paper, we suggest and implement a parallel large sparse linear system solver that improves performance on distributed memory systems like clusters by reducing processor idle times as much as possible by asynchronous iterations.


**키워드 : 비동기 알고리즘(asynchronous algorithm), 분산 메모리 시스템(distributed memory system), 초대형 희소 선형 시스템(large sparse linear system), 클러스터(cluster), MPI(Message Passing Interface)**

## 1. 서 론

오늘날 많은 자연과학 및 공학 문제들은 엄청난 양의 연산을 요구하므로 슈퍼컴퓨터의 사용이 필수적이다. 1990년대 초반부터 네트워크 기술의 발전과 저가의 고성능 프로세서의 등장으로, 고성능 컴퓨팅 분야는 점차 고가의 상용 슈퍼컴퓨터로부터 쉽게 제작이 가능한 저가의 클러스터로 옮겨가는 경향이 일어났다[1, 15].

병렬 프로그래밍 패러다임에 있어서는 여전히 동기 알고리즘(synchronous algorithm)이 주류를 이루는데, 올바른 연산을 위해서는 작업을 일찍 끝낸 프로세서들은 동기점(synchronization point)에서 데이터 교환을 위해 느린 프로세서들을 기다려야 하므로 부하 균형이 필수적이다[6]. 부하 불균형은, 때로는 비구조적 메쉬(unstructured mesh)처럼 문제의 성질에 따라 피할 수 없는 경우도 있으며, 비록 같은 크기로 작업을 나눈다 하더라도 이질적 클러스터(heterogeneous cluster)처럼 프로세서들의 성능이 다를 경우는 각기 연산시간이 달라진다.

이런 문제를 해결하는 하나의 방안으로 비동기 반복법(asynchronous iteration)의 개념이 나왔으며[4], 지금도 활발

※ 성능 실험을 위하여 Atom 클러스터를 독점 사용하도록 해주신 수원대학교 물리학과 이승종 교수님께 감사드린다. 한편 본 연구는 한국해양연구원이 수행하는 해양수산부 연구 과제의 지원으로 수행되었다.
† 정 회 원 : 수원대학교 컴퓨터과학과 교수
†† 정 회 원 : 삼성종합기술원 CSE Center
논문접수 : 2001년 7월 23일, 심사완료 : 2001년 9월 24일

히 연구되고 있다. 이는 알고리즘의 동기점을 가능한 한 제거함으로써 빠른 프로세서의 유휴 시간을 줄이는 것이 목적이다. 즉 비동기 알고리즘에서는, 각 프로세서는 다른 프로세서로부터 갱신된 데이터가 올 때까지 기다리지 않고 계속 다음 작업을 수행해 나간다. 따라서 동시에 갱신된 데이터를 교환한 후 다음 단계로 진행하는 동기 알고리즘에 비해, 미처 갱신되지 않은 데이터를 사용하는 경우가 많으므로 전체적으로는 연산량 대비의 수렴 속도는 느릴 수 있다. 그러나 각 프로세서는 거의 유휴 시간이 없이 연산을 수행하므로 wall clock time은 동기 알고리즘보다 적게 걸리며, 때로는 50%까지 빠른 결과도 보고되고 있다[3, 7].

그러나 현재까지 이루어진 모든 연구는 비동기 알고리즘의 구현이 쉬운 공유 메모리 컴퓨터를 사용하여 수행된 것으로, 분산 메모리 컴퓨터에서 구현한 연구는 알려져 있지 않다. 본 논문에서는 초대형 희소 선형 시스템의 해법에 국한해서, 동기점을 완전히 제거한 비동기 반복 알고리즘을 고안하여, 이를 분산 메모리를 가진 클러스터에 구현하고 동기 알고리즘과 성능을 비교하였다.

## 2. 선형 시스템의 비동기 반복해법 및 공유 메모리 시스템에서의 구현

chaotic relaxation이 Chazan & Miranker[4]에 의해 처음 제안된 이래 그 기법은 수치적인 문제를 다루는 데 있어서 여러 학자들에 의해 발전되어 왔다. Uresin & Dubois[17]는 이를 이산치 데이터 문제를 위한 비동기 반복법으로 일반화하고 그래프 문제와 같은 일반적인 전산학 문제들에도 적용할 수 있음을 보였으며 점차 사용 영역이 확대되고 있다[2, 5, 17]. Lu 등[11]은 비동기 병렬 스킴에 대한 좋은 개관을 제공한다.

본 논문에서는 초대형 희소 행렬 문제(large sparse matrix problem) $A\bar{x}=\bar{b}$($A$는 $n\times n$행렬, $b$는 길이 $n$의 벡터)를 푸는 비동기 반복 알고리즘에 국한하기로 한다. $L$개의 프로세서를 사용할 경우, $A\bar{x}=\bar{b}$는 다음과 같이 $L\times L$ 블록 형태(block form)로 쓸 수 있다.

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

그러면 원래의 문제는 $L$개의 작은 부분 문제들의 조합으로 나타낼 수 있다.

$$A_{kk}\overline{x_k} = \overline{b_k} - \sum_{j\neq k}^{L} A_{kj}\overline{x_j}, \quad k = 1, 2, \cdots, L. \quad (2)$$

이를 계산하는 일반적인 동기 병렬 알고리즘의 경우, 각 프로세서는 각자의 부분 해 $\overline{x_k}$를 계산한다. 그러나 (2)의 우변은 다른 프로세서가 계산하는 부분 해 $\overline{x_j}$ ($j\neq k$)와 연관되어 있으므로 전체적으로 수렴할 때까지 반복 연산하여야 한다. 따라서 각 프로세서는 자신이 갱신한 부분 해를 다른 프로세서와 교환하기 위해 프로세서 동기화는 필수적이다.

일반적으로 선형 시스템 $A\bar{x}=\bar{b}$의 해 $\bar{x}$의 근사치 $\bar{z}$의 수렴 여부는 잔차(residual) $\bar{r}\equiv\bar{b}-A\bar{z}$의 크기 $\|\bar{r}\|_2$로 판정한다. 문제를 여러 프로세서가 나누어 풀 경우, 부분 벡터 $\overline{x_k}$의 근사치 $\overline{z_k}$의 수렴 여부는 다음과 같이 정의된 부분 잔차의 크기로 판별할 수 있으며, $\|\bar{r}\|_2^2 = \sum_{k=1}^{L} \|\overline{r_k}\|_2^2$의 관계가 성립한다.

$$\overline{r_k} \equiv \overline{b_k} - A_{kk}\overline{z_k} - \sum_{j\neq k}^{L} A_{kj}\overline{z_j} \quad (3)$$

만일 프로세서의 수가 부분 문제의 수보다 적고 각 부분 문제를 1회 계산하는 데 걸리는 시간이 메시지 교환에 걸리는 시간에 비해 매우 크다면, 다음과 같이 프로세서 팜(processor farm) 모델의 변형된(즉 매스터가 따로이 존재하지 않는) 알고리즘을 사용할 수 있다. 공유 메모리 컴퓨터에서는 프로세서간의 메시지 교환에 시간이 거의 걸리지 않으므로 이런 모델이 적절한데, 비동기 알고리즘의 요소를 가미하면 다음과 같다[8, 16].

**Algorithm 1 (비동기 프로세서 팜 모델)**

1. 모든 프로세서는 초기치 벡터 $\overline{x_j}$($j=1, \cdots, L$)를 적절하게 선택한다.
2. 답이 만족스러울 때까지 각 프로세서들은 독립적으로 다음을 반복한다.
   1) 현재 다른 프로세서에 의해 계산되고 있지 않는 부분 문제 번호 $k$를 선택한다.
   2) $\overline{x_j}$($j\neq k$)를 공유 메모리로부터 읽는다.
   3) 부분 문제 $A_{kk}\overline{x_k} = \overline{b_k} - \sum_{j\neq k} A_{kj}\overline{x_j}$를 풀어 $\overline{x_k}$를 갱신하고 공유 메모리에 저장한다.
   4) 각 프로세서는 부분 잔차 $\overline{r_k}$를 계산한다.
   5) 하나의 프로세서가 부분 잔차를 종합함으로써 전체적인 수렴 여부를 판단한다.

동기 알고리즘과는 달리, 이 알고리즘의 단계 2.2)에서는 다른 프로세서가 갱신하는 $\overline{x_j}$($j\neq k$)의 값은, 갱신 여부와 상관없이 공유 메모리에 저장되어 있는 그대로 사용한다. 따

라서 빠른 프로세서는 느린 프로세서가 $\overline{x_j}$를 갱신하기를 기다릴 필요가 없으며, 느린 프로세서는 사용하려는 값이 빠른 프로세서에 의해 여러 번 갱신되더라도 가장 최근의 것만을 계산에 이용하게 된다.

한편 이런 비동기 알고리즘의 수렴에 필요한 조건은 다음과 같이 알려져 있다[16].

**정리 1.** 선형 시스템 $A\overline{x}=\overline{b}$에서, $A^{-1}$와 모든 대각 블록 $A_{ii}^{-1}$의 모든 성분들이 음이 아니며(nonnegative) 비대각 블록(off-diagonal block) $A_{lk}, l\neq k$ 의 모든 성분들이 양이 아니면(nonpositive) 어떤 초기 벡터 $\overline{x}^{(0)}$를 사용하여도 항상 옳은 해로 수렴한다.

## 3. 제안하는 동기 알고리즘

일반적으로 고성능 컴퓨팅 문제들은 엄청난 연산 시간과 메모리를 필요로 하므로 문제의 성질을 최대한 이용한 알고리즘을 사용할 수밖에 없다. 따라서 본 절에서는 우선 대상 문제를 푸는 특화된 동기 병렬 알고리즘을 고안하고, 4장에서는 이를 수정하여 비동기 알고리즘을 제안한다.

### 3.1 다루는 문제 및 성질

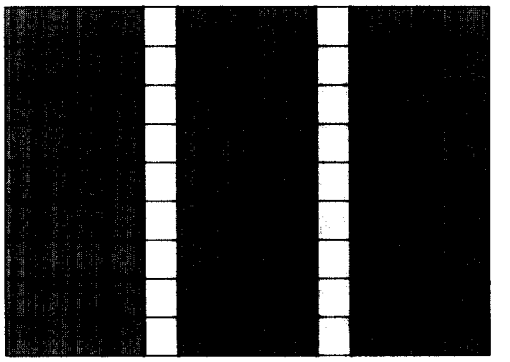
본 논문은 Kaufman[10]에 기술된 2큐 오버플로 큐잉 모델(overflow queuing model)을 인위적으로 약간 변형한 정칙 선형 시스템 $A\overline{x}=\overline{b}$을 다룬다. 근간이 되는 큐잉 문제의 파라미터는 $s_1=5, s_2=7, \lambda_1=10, \lambda_2=8, \mu_1=1, \mu_2=2, n_1=1{,}000, n_2=1{,}000$으로서 그 결과의 행렬 $Q$는 100만×100만인 거대한 희소행렬이다. 그러나 $Q$는 특이 행렬이므로 모든 대각 성분에 0.2를 더하여 정칙 행렬 $A\equiv Q+0.2I$로 만들고, 우변 벡터 $\overline{b}$는 그 행렬의 대각성분(즉 $b_i=a_{ii}$)을 취하였다. $Q$의 각 열의 합(column sum)은 0이므로 $A$는 열 방향으로 엄격 대각 우위(strictly diagonally dominant)의 성질을 갖는다. 따라서 비동기 반복 알고리즘의 수렴에 필요한 정리 1의 모든 조건을 만족하므로 정확한 해로의 수렴이 보장된다.

이를 (2)와 같이 부분 문제들로 나누어 풀 경우, 매 반복당 계산 시간은 짧은 반면, 각 프로세서가 동적으로 부분 문제를 선택하여 준비하는 데는 많은 시간이 걸린다. 그러나 모든 프로세서가 행렬 전체를 저장하여 사용하는 것은 메모리 제약상 불가능하며, 또한 클러스터에서는 프로세서간의 메시지 교환에 시간이 많이 걸리므로 Algorithm 1과 같은 프로세서 팜 모델은 부적절하다.

### 3.2 동기 알고리즘

본 논문에서는 데이터 병렬(data parallel) 모델을 사용하며 각 프로세서는 미리 정해진 부분 문제만을 풀도록 한다. 이 문제는 메쉬 문제는 아니나 큐의 각 상태가 4개의 다른 상태와 연관되어 있으므로 Park [13]처럼 가상의 2차원 메쉬 문제로 취급하여 영역분할법(domain decomposition)을 적용할 수 있다. 여러가지 방법으로 영역 분할하여 분할된 소영역을 각 프로세서에게 할당할 수 있으나, 1차원 분할을 사용하기로 한다. 예를 들어, (그림 1)은 15×10 메쉬를 3개의 프로세서에게 할당한 것으로, 각 프로세서는 빗금친 하나의 영역 내의 모든 격자점에서의 값을 계산하게 된다.

본 문제의 가상 메쉬의 크기는 1,000×1,000 인데, 라인 가우스-사이델법(line Gauss-Seidel method)[9]을 사용하여 하나의 수직 메쉬 라인 상의 모든 변수들의 값을 동시에 갱신한다. $i$ 번째 수직 메쉬 라인 상의 미지수들로 구성된 부분 벡터를 $\overline{x_i}$라 하자. (1)의 대각선 상의 행렬 $A_{ii}$는 삼대각 행렬(tridiagonal matrix)이므로 LU 분해를 이용하여 $\overline{x_i}$를 효율적으로 구할 수 있다.



(그림 1) 15×10 2차원 메쉬를 3개의 영역으로 나눈 예.

이 가상의 메쉬 문제를 행렬로 표현하면 100만×100만이 되므로 여러 개의 프로세서가 나눈다 하더라도 있는 그대로는 저장이 불가능하다. 그러나 문제의 행렬은 희소 행렬이므로 각 프로세서는 자신이 갱신하는 변수들과 관련된 부분 행렬만, 그것도 0이 아닌 성분만 5개의 벡터 형태로 가지면 된다. 즉 부분 벡터 $\overline{x_i}$를 갱신하는 프로세서는 단지 부분 행렬 $A_{i,i-1}, A_{i,i}, A_{i,i+1}$와 부분 벡터 $\overline{x_{i-1}}, \overline{x_i}, \overline{x_{i+1}}$만 가지면 된다.

각 수직 메쉬 라인 상의 미지수들로 구성된 부분 벡터 $\overline{x_i}$를 구하는 것이 하나의 부분 문제이다. 수직 메쉬 라인의 수는 1,000개이므로 원래의 문제를 1,000개의 부분 문제(즉 (1)의 $L$값은 1,000이다)로 나누고 각 프로세서에게 같은 수의 문

제를 할당한다. 예를 들어, $k$번째 프로세서에게 할당된 문제는 $k_1, k_2, \cdots, k_m$번째라 하고 그것들의 부분 해를 $\overline{x_{k_1}}, \cdots, \overline{x_{k_m}}$이라 하자. 한편, 2차원 메쉬 상에서 프로세서 $k$가 갱신하는 영역의 왼편 및 오른편 영역의 문제를 계산하는 프로세서들을 편의상 각기 좌측 및 우측 프로세서라고 부르기로 한다. 각 부분 문제를 LU 분해법으로 계산한다면 다음과 같은 동기 알고리즘을 사용할 수 있다.

**Algorithm SYNC ($k$번째 프로세서의 동기 알고리즘)**

1. 초기치 벡터 $\overline{x_{k_1-1}}, \overline{x_{k_1}}, \cdots, \overline{x_{k_m}}, \overline{x_{k_m+1}}$을 적절하게 선택한다.
2. 대각 블록들 $A_{k_1k_1}, \cdots, A_{k_mk_m}$을 LU 분해한다.
3. 매스터 프로세서로부터 계산을 끝내라는 신호가 올 때까지 다음을 반복한다.
   1) $\overline{x_{k_1}}, \cdots, \overline{x_{k_m}}$을 하나씩 순서대로 업데이트한다.
   2) $\overline{x_{k_1}}$과 $\overline{x_{k_m}}$을 각기 좌측 및 우측의 인접 프로세서에게 보내며, $\overline{x_{k_1-1}}$과 $\overline{x_{k_m+1}}$을 각기 좌측 및 우측의 인접 프로세서로부터 받는다.
   3) 부분 잔차 크기의 제곱 $\sum_{j=1}^{m} \| \overline{r_{k_j}} \|_2^2$를 계산하여 매스터 프로세서에게 보낸다.
   4) 매스터 프로세서는 부분 잔차를 모아 전체 잔차 $\overline{r}$의 $\| \overline{r} \|_2^2$를 계산한다.
   5) 만일 $\| \overline{r} \|_2^2$의 값이 만족스러우면 매스터 프로세서는 다른 프로세서에게 계산을 끝내라는 신호를 보낸다.

매스터 프로세서는 슬레이브로부터의 부분 잔차의 정보를 모아 수렴 여부를 판단하며 계산을 언제 종료할 것인가를 결정하여 전체 슬레이브에게 통보하는 역할을 한다. 따라서 슬레이브에 비해 하는 일이 너무 적으므로 매스터를 따로 둘 필요가 없고 슬레이브 프로세서 중 하나가 자신의 계산 도중 틈틈이 그 역할을 담당하도록 한다.

이 알고리즘에는 모든 프로세서가 데이터를 교환하기 위한 단계 3.2)와 전체적인 계산의 수렴 판단을 위한 3.5)의 두 군데 동기점이 존재한다.

## 4. 제안하는 비동기 알고리즘

### 4.1 수렴 판단을 위한 동기점의 제거

Algorithm SYNC의 3.5)와 같이 수렴 판단을 위한 동기점은 일반적인 비동기 프로세서 팜 모델 Algorithm 1의 단계 2.5)에도 존재한다. 다음과 같이 비동기 잔차를 정의하고 이용함으로써 이런 동기점을 제거할 수 있다.

**정의 1.** $k_l$번째 부분 벡터 $\overline{x_{k_l}}$의 새로이 갱신된 근사치를 $\overline{z_{k_l}}$하자. 그러면 $\overline{z_{k_l}}$의 비동기 부분 잔차는 다음과 같이 정의한다.

$$\overline{r_{k_l}^{(?)}} = \overline{b_{k_l}} - A_{k_l}\overline{z_{k_l}} - \sum_{j \neq k_l} A_{k_l,j}\,\overline{z_j^{(?)}}$$

여기서 $\overline{z_j^{(?)}}$는 자신 및 다른 프로세서에 의해 계산된 $\overline{x_j}$의 근사치로서 "갱신의 여부와 상관없이 현재의 값 그대로"를 의미한다.

만일 $\overline{z_j^{(?)}}$의 값이 모두 갱신된 것이라면 $\overline{r_{k_l}^{(?)}}$는 (3)에서 정의된 동기 부분 잔차 $\overline{r_{k_l}}$와 일치한다. $k$번째 프로세서는 $\overline{x_{k_1}}, \cdots, \overline{x_{k_m}}$을 갱신하고 이들의 부분 잔차 크기의 제곱을 계산하여 매스터 프로세서에게 보낸다. 한편 매스터 프로세서는 각 프로세서로부터 비동기적으로 그 값을 받으므로 전체 잔차의 추정치 $\| \overline{r} \|_2^2$ 역시 비동기 잔차가 된다.

어느 시점의 비동기 잔차는 각 부분 벡터별로 다른 수의 반복 해법을 적용한 후 동기 잔차를 계산하는 것과 동일하다. 그러므로 엄밀히는 동기 잔차와 약간 다르나 이론적으로 거의 차이가 없으며, 실제 비동기 잔차는 충분히 좋은 수렴 판단의 근거가 된다.

### 4.2 데이터 교환을 위한 동기점의 제거에 따른 문제

Algorithm 1처럼 공유 메모리 시스템에서 이를 구현하는 것은 아주 쉽다. 즉 단순히 각 프로세서가 독립적으로 갱신한 값을 공유 메모리에 저장하고, 필요할 때 원하는 값을 읽도록 하면 결과적으로 비동기적으로 메시지를 교환하는 셈이 된다.

Message Passing Interface(MPI)[12]는 병렬 프로그래밍을 위한 메시지 패싱 라이브러리의 표준이다. 그러나 분산 메모리 시스템에서 MPI를 사용하여 비동기 알고리즘을 구현하는 것은 간단하지 않고 다음의 두가지 문제를 일으키는 사실이 발견되었다.

첫째, 동기점을 제거하여 각 프로세서가 임의로 메시지를 보내면, 빠른(혹은 부하가 작은) 프로세서는 계산을 빨리 끝내고 자주 메시지를 보내는 반면, 느린(혹은 부하가 큰) 프로세서는 이를 미처 처리하지 못해 결국 시스템 버퍼 오버플로로 발전하였다. 그 이유는, MPI는 올바른 연산을 위해 보내는 메시지의 수와 받는 메시지의 수가 같도록 요구하기 때문이다.

둘째, 동기 알고리즘과는 달리 비동기적 연산에서는, 각 프로세서는 어느 시점에 어느 프로세서가 어떤 종류의 메시지

를 보내올 지 알 수 없다는 점이다.

### 4.3 버퍼 오버플로 및 비동기적으로 전달되는 메시지의 처리 방법

버퍼 오버플로 문제는 별도의 서버 프로세서를 두어 메시지 교환을 전담하도록 하여 해결하였다. 즉 Algorithm SYNC와는 달리 각 프로세서(즉 클라이언트)는 데이터를 서버에게만 보내도록 하였다. 서버는 항상 클라이언트로부터 오는 메시지가 있는지 확인하고 있으면 즉시 수신함으로써 버퍼를 비우고 클라이언트의 대기 시간을 줄인다. 새로 도착한 메시지가 있는지의 여부는 MPI 함수 MPI_IProbe( )를 사용하였다.

두 번째 문제는 MPI_IProbe( )의 인자의 값을 적절히 이용함으로써 해결하였다. 일반적으로 MPI 함수들은 메시지의 소스(source)와 종류를 구분하는 택(tag)을 지정해야 한다. 따라서 MPI_IProbe( )의 인자로서 MPI_ANY_SOURCE와 MPI_ANY_TAG를 사용하여 일단 메시지를 수신하고 구조체 status의 status.MPI_SOURCE로 발신자를, status.MPI_TAG로 메시지의 종류를 파악하고 메시지의 종류에 따라 적절한 작업을 수행하도록 하였다.

이상의 문제들을 해결하는 핵심 루틴은 다음과 같다. requester와 jobkind에 따라 적절히 수행되어야 하는 작업은 다음의 Algorithm SVR에 설명되어 있다.

```
do {
   MPI_IProbe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm,
   &flag, &status) ;
   If (flag) {    /* 도착한 메시지가 있으면 flag은 TRUE */
      requeste r = status.MPI_SOURCE ;
      jobkind = status.MPI_TAG ;
      (requester와 jobkind에 따라 적절한 작업을 수행함).
   }
} while (flag) ;
```

### 4.4 제안하는 비동기 반복 알고리즘

클라이언트의 알고리즘은 Algorithm SYNC와 동일하며 단지 단계 3.2)와 3.4)에서 서버와 데이터를 교환하는 점만이 다르다. 그러나 조금이라도 빨리 인접 프로세서가 필요로 하는 결과를 서버에게 보내기 위해 단계 3.1)과 3.2)에서 다음과 같이 연산의 순서를 바꾸었다.

3.1)′ $\overline{x_{k_1}}$과 $\overline{x_{k_m}}$을 갱신하여 서버에게 보낸다.

3.2)′ $\overline{x_{k_2}}, \cdots, \overline{x_{k_{m-1}}}$을 계산하고 $\overline{x_{k_1-1}}$과 $\overline{x_{k_m+1}}$을 서버로부터 받는다.

클라이언트가 갱신한 $\overline{x_{k_1}}$과 $\overline{x_{k_m}}$을 서버에게 보내면 서버는 즉시 $\overline{x_{k_1-1}}$과 $\overline{x_{k_m+1}}$를 보내며, 부분 잔차를 보내면 서버는 즉시 전체 잔차를 계산해 수렴 여부를 판단하고 계산 종료의 여부를 결정하여 클라이언트에게 알려준다. 따라서 클라이언트는 MPI_IProbe( )를 사용할 필요가 없다.

서버는 메시지 패싱의 핵심적 역할을 담당하는데 그 알고리즘은 다음과 같다.

**Algorithm SVR (서버 프로세서)**

답이 충분히 수렴할 때까지 다음을 반복한다.

1. 클라이언트로부터 오는 메시지를 탐지한다. 만일 그것이 $k$번째 클라이언트이면

   Case 1. 만일 갱신된 데이터(즉 $\overline{x_{k_1}}$과 $\overline{x_{k_m}}$)가 오면

   1) 지역 메모리에 그 값을 저장하고
   2) 그 클라이언트가 필요로 하는 경계치 $\overline{x_{k_1-1}}$과 $\overline{x_{k_m+1}}$를 보낸다.

   Case 2. 만일 부분 잔차의 값이 오면

   1) 지역 메모리에 그 값을 저장하고
   2) 모든 부분 잔차의 값을 더하여 전체의 비동기 잔차를 계산하여 수렴 여부를 확인한다.
   3) 만일 충분히 수렴했으면 그 클라이언트에게 작업 중지 명령을 내린다.

그러나 서버가 하는 일은 클라이언트에 비해 극히 적으므로 서버 프로세서의 효율적 사용을 위해 독립된 서버를 두지 않고 클라이언트 중의 하나가 틈틈이 서버 역할도 수행하도록 하였다. 이 모델(**Algorithm ASYNC**라 하자)은 앞서 말한 클라이언트-서버 모델이 변형이므로 별도로 기술하지는 않는다. 따라서 클라이언트와 서버(또는 서버 역할을 수행하는) 사이는 동기화가 되나 전체적으로 클라이언트들은 다른 것들의 작업 진행 상황과 무관하게 계산을 수행하므로 비동기적이다.

## 5. 성능 실험 및 고찰

Algorithm SYNC와 ASYNC의 비교 실험에는 수원대학교의 Atom 클러스터를 독점 사용하였다. 이는 Beowulf 클러스터[14]로서 <표 1>처럼 이질적 클러스터이나 사용자는 원하는 프로세서만을 지정하여 사용할 수 있다. 노드들은 100Mbps 스위칭 허브로 연결되어, 노드 사이의 메시지 패싱에 걸리는 시간은 모두 동일하다.

<표 2>는 비동기 Algorithm ASYNC를 동기 Algorithm SYNC와 비교한 것으로, CPU의 종류와 상관없이 총 10개의 프로세서에게 각기 100개씩의 수직 메쉬 라인을 할당하여

잔차가 $2\times10^{-9}$에 달할 때까지 걸린 wall clock time 및 그때까지 각 프로세서가 수행한 반복 회수를 나타낸다. "결과명"에서 S는 SYNC, A는 ASYNC의 결과를 나타내며, 그 다음의 숫자 중 하이픈 왼편은 Celeron의 CPU 개수, 오른쪽은 Pentium 3(P3)의 개수를 나타낸다. 단 곱셈 형태로 숫자가 주어진 것, 예를 들어 $2\times m+n$은, $m$개 노드의 CPU는 모두(즉 둘 다), 다른 $n$개의 노드에서는 CPU를 하나씩 사용한 것을 의미한다. 두 종류의 CPU를 함께 사용한 경우, 서버 역할은 반드시 P3 CPU가 담당하였다.

〈표 1〉 Atom 클러스터의 구성

| | 대 | CPU | main memory |
|---|---|---|---|
| master | 1 | single 700MHz Pentium 3 | 256MB |
| slave | 7 | dual 466MHz Celeron | 128MB |
| | 15 | dual 700MHz Pentium 3 | 256MB |

〈표 2〉 Algorithm SYNC와 ASYNC의 비교

| 결과명 | 사용 CPU | | wall clock time(sec) | 수렴할 때까지의 반복 회수 | | |
|---|---|---|---|---|---|---|
| | Celeron | P3 | | server | server의 Celeron | server의 P3 |
| S5-0 | 5 | 0 | 259.21 | 885 | | |
| A5-0 | | | 236.84 | 832 | 830~832 | - |
| S0-5 | 0 | 5 | 182.83 | 885 | | |
| A0-5 | | | 176.38 | 831 | 831~832 | - |
| S5x2-0 | 5×2 | 0 | 165.58 | 891 | | |
| A5x2-0 | | | 158.39 | 823 | 824~908 | - |
| A9-1 | 2×2+5 | 1 | 148.66 | 1268 | 794~1083 | - |
| A8-2 | 2×1+6 | 2 | 125.94 | 1066 | 681~912 | 1211 |
| A7-3 | 7 | 3 | 117.54 | 984 | 792~853 | 1111, 1131 |
| A6-4 | 6 | 4 | 113.27 | 938 | 767~824 | 1071~1091 |
| S5-5 | 5 | 5 | 131.40 | 891 | | |
| A5-5 | | | 108.56 | 900 | 741~791 | 1027~1045 |
| A4-6 | 4 | 6 | 103.97 | 850 | 714~758 | 984~997 |
| A3-7 | 3 | 7 | 99.59 | 811 | 674~723 | 946~952 |
| A2-8 | 2 | 8 | 94.35 | 765 | 655, 688 | 894~905 |
| A1-9 | 1 | 9 | 90.27 | 729 | 655 | 852~861 |
| S0-10 | 0 | 10 | 94.09 | 891 | | |
| A0-10 | | | 86.88 | 695 | - | 819~831 |
| S0-5x2 | 0 | 5×2 | 116.73 | 891 | | |
| A0-5x2 | | | 108.67 | 734 | - | 814~830 |

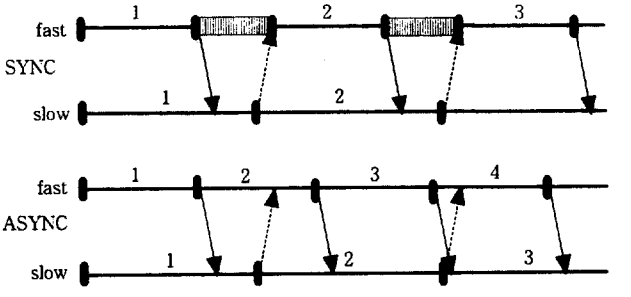
Algorithm SYNC와 ASYNC를, 같은 종류의 CPU만을 사용(따라서 균질 클러스터와 동일)한 결과들(즉 사용한 CPU 개수가 5-0, 0-5, 5x2-0, 0-5x2인 것들)을 비교해 보면 모든 경우 ASYNC가 SYNC보다 약 3.5%~8.4% 정도 시간이 적게 걸린 것을 알 수 있다.

Celeron과 P3 CPU를 각기 5개씩 사용한 결과를 비교하면 ASYNC는 SYNC보다 시간이 약 17.5% 정도 감소했고, Celeron이 수렴에 이르기까지 741~791회 반복 연산한 반면, P3는 1027~1045회 반복하여 느린 Celeron의 연산을 가속화했음을 알 수 있다. 한편 P3인 서버는 클라이언트들과의 메시지 패싱으로 인하여 900회 반복 연산하는 데 그쳤다.

나머지는 Celeron : P3의 CPU의 개수를 변화시키며 계산한 결과를 보여준다. 당연히 P3의 개수가 늘어날수록 시간은 감소하므로 서버, Celeron, P3 모두 반복 회수가 줄어든다.

한 가지 특기할 점은, 같은 비율의 Celeron : P3를 사용하더라도 같은 노드의 CPU를 둘 다 사용하는 경우(예를 들어 0-5x2)는 각기 다른 노드들로부터 CPU를 하나씩만 사용하는 경우(예를 들어 0 ~10)에 비해 SYNC이든 ASYNC이든 약 25% 정도 속도 저하 현상이 발견되었다. 이는 같은 노드 내에서 두 개의 CPU에 의한 메인 메모리, LAN 카드 등 모든 하드웨어의 공유에 의한 속도 저하, SMP를 지원하는 Linux 운영체제의 비효율성 등이 그 원인이 될 수 있을 것이다.

(그림 2)는 Algorithm SYNC와 ASYNC에 있어서 성능이 3 : 2인 두 프로세서의 시간에 따른 연산 과정을 나타낸다. 화살표는 다른 프로세서에게 보내는 메시지를 표시하며 숫자는 연산의 번호를 나타낸다. Algorithm SYNC의 경우, 빗금 친 부분은 빠른 프로세서가 자신이 필요한 데이터가 오기를 기다리는 시간으로 전체적인 속도는 느린 프로세서의 성능에 의해 제약됨을 알 수 있다.




(그림 2) Algorithm SYNC와 ASYNC에서 빠른 프로세서와 느린 프로세서의 시간에 따른 연산 과정

Algorithm ASYNC의 경우, 빠른 프로세서는 2번째 및 4

번째 연산을 시작할 때까지 느린 프로세서로부터 갱신된 값을 받지 못하고 이전의 값을 그대로 사용하며, 느린 프로세서는 3번째 연산 도중에 빠른 프로세서로부터 2개의 갱신된 값이 전달되나 늦게 수신된 최신 값만을 사용해 4번째 연산을 시작하게 된다.

실제로는 클라이언트 중의 하나가 서버의 역할을 겸하므로 실제는 약간의 유휴 시간이 있을 것이나, 빠른 프로세서도 거의 유휴 시간이 없이 작업을 하므로 시간이 적게 걸릴 것임이 설명된다. 즉 빠른 프로세서는 동기 알고리즘에 비해 더 많은 회수의 반복 연산을 하므로 더 나은 결과를 얻게 되고, 그 결과 느린 프로세서는 동기 알고리즘에 비해 더 정확한 데이터를 받으므로 느린 프로세서의 수렴 속도도 더 빨라지는 것이다.

## 6. 결 론

네트워크 기술의 발전과 저가의 고성능 프로세서의 등장으로 쉽게 제작이 가능하며 가격 대 성능비가 뛰어난 클러스터가 점차 각광을 받고 있다. 특히 인터넷 서비스 분야에서 소수의 고성능 서버대신 부하 분산 및 웹 서버로서 클러스터가 많이 사용되고 있다.

그러나 초대형 자연과학 및 공학 연산에서 클러스터는 분산 메모리 시스템이므로 공유 메모리 시스템에 비해서는 프로그래밍이 어려운 점이 제약사항이나 점차 MPI를 사용해 프로그램을 병렬화 해나가고 있는 실정이다. 그러나 아직 대부분의 병렬 알고리즘은 동기 알고리즘으로 올바른 계산을 위해서는 프로세서들의 동기화와 부하균형이 필수적인데, 전체적인 성능은 가장 느린(또는 부하가 큰) 프로세서에 의해 결정된다.

본 논문에서는 100만×100만 크기의 초대형 선형 시스템을 푸는 동기 병렬 알고리즘을 고안하고 이를 클러스터 상에서 구현하였다. 또한 여기에 비동기 반복법의 개념을 도입하여 빠른 프로세서의 유휴 시간을 줄임으로써 전체적으로 성능을 향상시키는 비동기 병렬 알고리즘을 제안하고 성능 실험을 통하여 동기 알고리즘과 비교하였다. 실험 결과 부하가 잘 분산된 동질의 프로세서만 사용할 경우에도 비동기 알고리즘이 3.5%~8.4% 정도 시간이 적게 걸렸으며, 이질적인 클러스터의 경우 거의 30% 정도의 성능 향상을 보였다.

본 성능 실험에서는 Atom 클러스터 전체를 독점 사용하였으나 대부분의 경우 여러 사람이 공유해 사용하므로 독점 사용이 쉽지 않다. 따라서 제안한 비동기 알고리즘은 많은 사람이 공유하는 동질적 혹은 이질적 클러스터, 그리고 LAN 또는 Internet 상에 연결된 다수의 컴퓨터를 사용하여 하나의 거대한 문제를 푸는 경우, 프로세서들의 서로 다른 성능과 연산 이외의 부하에 의해 전체적인 계산 수행속도가 떨어지는 것을 완화하는 데 효과적으로 사용될 수 있다.

## 참 고 문 헌


[1] M. Baker and R. Buyya. "Cluster computing at a glance," in High performance cluster computing : Architecture and systems, Vol.1, R. Buyya, ed., Prentice Hall, 1999.

[2] B. Barán, 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] D. Chazan and W. Miranker, "Chaotic relaxation," Linear Algebra and Its Applications, Vol.2, pp.199-222, 1969.

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

[8] A. Frommer and D. B. Szyld, "On asynchronous iterations," Research Report 99-5-31, Department of Mathematics, Temple University, 1999.

[9] G. H. Golub and C. F. Van Loan, "Matrix computations," 3rd Ed., Johns Hopkins University Press, Baltimore, 1996.

[10] L. Kaufman, "Matrix methods for queuing problems," SIAM J. Sci. Stat. Comput., Vol.4, pp.525-552, 1983.

[11] E. J. Lu, M. G. Hilgers, and B. McMillin. "Asynchronous parallel schemes : A survey," Technical Report CSC 93-19, Dept. of Computer Science, University of Missouri-Rolla, Nov. 1993.

[12] MPI Forum, "MPI : A Message-Passing Interface standard," 1995.

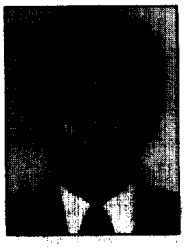
[13] P. S. Park, "A domain decomposition method applied to queuing network problems," Comm. Kor. Math. Soc., Vol.10, pp.735-750, 1995.

[14] Scyld Computing Corporation, "The Beowulf Project," http ://www.beowulf.org/, 2001.

[15] L. M. Silva and R. Buyya, "Parallel programming models and paradigms," in High performance cluster computing : Programming and application issues, Vol.2, R. Buyya, ed., Prentice Hall, 2000.

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

[17] 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@mail.suwon.ac.kr

1977년 서울대학교 해양학과 졸업(학사)

1978년~1982년 KIST 부설 해양연구소 연구원

1984년 미국 올드 도미니언 주립대학 계산 및 응용수학과 석사

1991년 미국 메릴랜드 주립대학 응용수학과 박사

1991년~1995년 한국해양연구소 선임연구원(전산실장, 수치모델 연구그룹장 역임)

1995년~현재 수원대학교 컴퓨터과학과 조교수

관심분야 : 수치 선형대수, 고성능 컴퓨팅(특히 수치 계산분야), 큐잉 문제의 수치해법



## 신 순 철

e-mail : scshin@sait.samsung.co.kr

1998년 수원대학교 전자계산학과 졸업 (학사)

1999년~2000년 한국해양연구원 위촉연구원

2001년 수원대학교 전자계산학과 졸업 (석사)

2001년~현재 삼성종합기술원 CSE Center 연구원

관심분야 : 고성능 컴퓨팅, 시스템 통합(자원 공유)