분산 메모리 시스템에서의 MPMD 방식의 비동기 반복 알고리즘을 위한 비대칭 전송의 구현

Implementation Of Asymmetric Communication For Asynchronous Iteration By the MPMD Method On Distributed Memory Systems

박 필 성*
Pil-Seong Park

요 약

비동기 반복 알고리즘은 부하 불균형 및 컴퓨터 노드 간의 전송 지연에 의한 병렬 알고리즘의 성능 저하를 완화하는 하나의 방법인데, 이는 노드들 간의 비대칭적 데이터 전송을 필요로 한다. 본 논문에서는 분산 메모리 시스템 상에서 MPMD 방식으로 노드당 별도의 서버 프로세스를 추가로 생성하여 비대칭적 전송을 구현하고, 노드당 하나의 프로세스를 생성하는 SPMD 방식과 비교하며 그 장단점에 대해 논의한다.

Abstract

Asynchronous iteration is a way to reduce performance degradation of some parallel algorithms due to load imbalance or transmission delay between computing nodes, which requires asymmetric communication between the nodes of different speeds. To implement such asynchronous communication on distributed memory systems, we suggest an MPMD method that creates an additional separate server process on each computing node, and compare it with an SPMD method that creates a single process per node.

Keyword : asynchronous iteration, asymmetric communication, MPI(Message Passing Interface), SPMD(single program multiple data), MPMD(multiple program multiple data)

1. 서 론

오늘날 하나의 수퍼컴퓨터로는 불가능한 방대한 계산을 필요로 하는 새로운 형태의 응용 프로그램들이 등장하고 있는데, 이들은 지리적으로 분산된 GRID 환경을 이용함으로써 효율적으로 실행시킬 수 있다 [8]. 그러나 분산 컴퓨팅 환경은 기존의 단일 병렬 시스템과는 많은 점에서 다르며 이전의 기술들을 그대로 적용하기에는 한계가 있다. 기존 병렬 연산의 주류는 동기 알고리즘(synchronous algorithm)으로, 동기화(synchronization)가 필요하며 부하 균형이 필수적이다[7]. 그러나 문제에 따라 부하 균형이 어렵거나, 이질 클러스터(heterogeneous cluster)처럼 프로세서들의 성능이 서로 다르거나, 지리적으로 분산된 환경에서는 네트워크 전송 지연 등으로 유휴시간이 길어질 수밖에 없다.

비동기 반복(asynchronous iteration) 알고리즘은 유휴 시간으로 인한 성능 저하를 완화하는 하나의 방법으로, 각 프로세서는 유휴시간이 거의 없이 자신의 작업을 수행하며 다른 프로세서로부터 데이터가 오지 않으면 이전의 데이터를 사용하여 계산을 계속 수행해 나간다. 따라서 동기점에서 최신의 데이터를 교환하고 계산을 진행하는 동기 알고리즘에 비해 연산량 대비의 수렴 속도는 느리다. 그러나 유휴시간이 거의 없으므로 프로그램 수행의 실제 시간(wall clock time)은 동기 알고리즘

* 종신회원 : 수원대학교 컴퓨터학과 부교수
pspark@suwon.ac.kr(제1저자)

보다 1/3-1/2까지 단축된 결과도 보고되고 있다[3].

비동기 반복 알고리즘의 구현을 위해서는, 송신과 수신 회수의 관점에서 비대칭적 전송이 필요하다. 즉 빠른 프로세서가 보내는 많은 데이터를 느린 프로세서는 "무시"하고, 빠른 프로세서는 느린 프로세서의 수신 여부와 무관하게 데이터를 계속 보낼 수 있어야 한다. 클러스터와 같은 분산 메모리 환경에서는 명시적인 메시지 교환을 통해 데이터의 교환이 이루어지는데, 모든 프로토콜들은 올바른 전송을 위해 당연히 송신과 수신이 1:1로 매칭되는 것을 요구하므로 있는 그대로는 비대칭적 송수신의 구현이 불가능하다.

CPU당 하나의 프로세스만을 생성하여 비대칭적 송수신을 구현하는 경우(예를 들면 [1]의 방법), 그 프로세스는 CPU 시간을 나누어 두 가지 기능(즉 본연의 연산 기능 및 다른 프로세스의 대기 시간을 줄이도록 즉시 메시지를 수신하는 기능)을 겸할 수밖에 없는데, 한 기능의 향상은 다른 기능의 저하를 가져온다. 따라서 본 논문에서는 연산과 메시지 수신 부분을 분리하여 같은 CPU상의 각기 다른 프로세스가 담당하게 하는 MPMD(multiple program multiple data)방식을 제안하고 비교하기로 한다.

2. 비동기 반복 알고리즘 및 MPI

2.1 비동기 반복 알고리즘

Chazan & Miranker [5]에 의해 제안된 이래, 비동기 반복 알고리즘은 많은 연구가 이루어졌으며 [9,12], 점차 일반적인 전산학 문제로 응용 영역이 확대되고 있다[2,6,14]. 본 논문의 목적은 이미 보고된 방법보다 더 효율적인 비동기 반복 알고리즘을 고안하는 것이 아니라, MPMD방식으로 비대칭 전송을 쉽게 구현하고자 하는 것이다. 따라서 선형 시스템의 비동기 반복법 중 가장 간단한 다음의 블록 자코비(block Jacobi) 방법을 예로 들기로 하나, 제안하는 비대칭 전송 방법은 비동기 반복법의 다른 모델에서도 적용이 가능하다.

선형 시스템 $A\bar{x}=\bar{b}$(A는 $n\times n$, $\bar{x}$와 $\bar{b}$는 길이 n 인 벡터)을 L개의 작은 문제로 나누는 경우 다음과 같이 블록 형태로 쓸 수 있다. 여기서 대각 블록 A_{ll}($l=1,\ldots,L$)은 정방행렬이며 벡터 $\bar{x}$와 $\bar{b}$는 A의 소블록들의 크기에 맞도록 파티션 되었다고 가정한다.

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

그러면 (1)은 다음과 같이 L개의 작은 부분 문제들의 집합으로 나타낼 수 있다.

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

이를 계산하는 일반적인 동기 병렬 알고리즘의 경우, 각 부분 벡터 $\overline{x_k}$는 여러 개의 프로세서에게 나누어져 계산된다. 그러나 (2)의 우변은 다른 프로세서가 계산하는 부분 벡터와 연관되어 있으므로 전체적으로 수렴할 때까지 반복 연산해야 한다. 따라서 각 프로세서는 자신이 계산한 부분 벡터를 다른 프로세서와 교환해야 한다. 프로세서의 수가 부분 문제의 수 L보다 적을 경우 공유 메모리 시스템에 적용한 알고리즘은 다음과 같다 [13].

Algorithm 1 (processor farm model on distributed memory systems)

1. 부분 벡터 $\overline{x_1}, \overline{x_2}, \ldots, \overline{x_L}$의 초기치를 적절하게 택하고 저장한다.
2. 모든 프로세서는 답이 만족스러울 때까지 다음을 독립적으로 반복 시행한다.
   1) 다른 프로세서에 의해 계산되고 있지 않은 l (부분 문제의 번호)을 선택한다.
   2) $\overline{x_k}$($k\neq l$)를 공유 메모리로부터 읽는다.
   3) $A_{ll}\overline{x_l} = \overline{b_l} - \sum_{k\neq l} A_{lk}\overline{x_k}$을 풀어 $\overline{x_l}$을 계산한다.

4) $\overline{x_l}$을 공유 메모리에 저장한다.

2.2 MPI

메시지 패싱(message passing)은 프로세서간에 교환할 데이터를 메시지 전달 함수를 이용하여 메시지 형태로 주고받는 병렬 연산 모델이다. 메시지 패싱 라이브러리(message passing library)는 이런 함수들의 집합체로서 MPI(Message Passing Interface)가 표준으로 제정되었다. MPI 라이브러리는 각 벤더들의 상용 라이브러리 외에 LAM, MPICH, WMPI, MacMPI 등 여러 종류가 있다. 본 논문에서 사용할 LAM 환경에서는 계산 주체가 CPU가 아니라 프로세스이며 프로그램 실행시 필요한 프로세스의 개수를 지정하게 된다. 각 프로세스는 랭크(rank)에 의해 구분되며, n개의 프로세스를 사용할 경우 0부터 n-1의 값을 가진다.

어느 프로세스가 송신함수가 호출되면 필히 수신함수를 호출하는 프로세스가 있어야 하며, 이 경우 송수신 데이터의 수, 데이터의 형, 택(tag) 등이 일치해야 하고 송수신의 상대 프로세스는 상대방의 랭크를 사용하여 지정한다.

MPI의 데이터 전송 모드는 버퍼의 사용 여부 등에 따라 standard, synchronous, ready, buffered 모드의 4가지가 있다[11]. 비록 standard 모드나 buffered 모드에서 버퍼를 사용한 비동기적인 전송 방법을 제공하고 있으나, 이는 시간적으로 지연시킬 뿐이며 궁극적으로 버퍼 내에 일시적으로 저장된 모든 데이터는 같은 회수의 수신 함수 호출에 의해 하나씩 처리되어야 하므로 있는 그대로는 비대칭적 송수신의 구현이 불가능하다.

3. SPMD 방식의 비대칭 송수신의 구현

3.1 문제 및 동기 병렬 알고리즘

본 논문은 [1]에서 사용한 것과 유사한 선형 시스템 $A\overline{x} = \overline{b}$의 해법을 다룬다. A는 1,600만×1,600만인 거대 희소행렬이며, $\overline{x}$와 $\overline{b}$는 길이가 각기 1,600만인 벡터들이다. A의 그래프를 그리면 4,000×4,000의 2차원의 격자가 되므로 [1], 전체 벡터는 길이가 각기 4,000인 L=4,000개의 각 부분 벡터로 나누도록 한다. 각 부분 벡터들은 수직 격자선 상의 값을 나타내는데, 부분벡터 $\overline{x_l}$은 $\overline{x_{l-1}}$ 및 $\overline{x_{l+1}}$만 연관이 있고($l=1$의 경우는 $\overline{x_2}$만, $l=L$의 경우는 $\overline{x_{L-1}}$만) 다른 부분벡터는 관련이 없다.

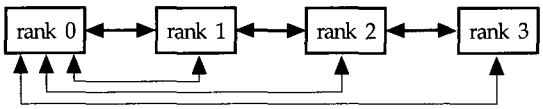
따라서 L=4,000개의 부분 문제(식 (2) 참조)로 나누어, 각 프로세스가 여러 개의 부분 벡터 연산을 수행하는 동기 병렬 알고리즘은 다음과 같다. 모든 프로세스는 직접 상호 메시지 교환이 가능하며, k번째 프로세스는 부분벡터 $\overline{x_{k_1}}, \ldots, \overline{x_{k_m}}$의 계산을 수행한다고 가정한다.

Algorithm 2(k번째 프로세스의 동기 알고리즘)

1. 부분 벡터 $\overline{x_{k_1-1}}, \overline{x_{k_1}}, \ldots, \overline{x_{k_m}}, \overline{x_{k_m+1}}$의 초기치를 적절하게 택하고 대각 블록들 $A_{k_1k_1}, \ldots, A_{k_mk_m}$을 LU 분해한다.
2. 매스터 프로세스로부터 STOP 신호가 올 때까지 다음을 반복한다.
   1) $\overline{x_{k_1}}, \ldots, \overline{x_{k_m}}$을 하나씩 순서대로 업데이트한다.
   2) 경계치 $\overline{x_{k_1}}$과 $\overline{x_{k_m}}$을 각기 좌측 및 우측의 인접 프로세스에게 보낸다.
   3) $\overline{x_{k_1-1}}$과 $\overline{x_{k_m+1}}$을 각기 좌측 및 우측의 인접 프로세스로부터 받는다.
   4) 부분 잔차 크기의 제곱 $\sum_{j=1}^{m} \| \overline{r_{k_j}} \|_2^2$를 계산하여 매스터 프로세스에게 보낸다.
   5) 매스터 프로세스는 4)를 사용하여 전체 잔차 $\overline{r}$의 $\| \overline{r} \|_2^2$를 계산하고 그 크기가 충분히 작으면 모든 프로세스에게 STOP 신호를 보낸다.

매스터 프로세스는 다른 프로세스들처럼 연산에 참여하나, 2.5처럼 전체 잔차의 크기를 계산하고 계산의 계속 여부를 결정하는 역할을 겸한다. 예

를 들어 4개의 프로세스를 사용해 계산하는 경우, rank 0을 매스터 프로세스로 지정하면, 각 프로세스간의 데이터 교환은 그림 1과 같다. 굵은 화살표는 2.2 및 2.3에서의 부분 벡터 데이터의 교환을 나타내며, 가는 화살표는 2.4 및 2.5의 메시지 전송을 나타낸다.



(그림 1) 4개의 프로세스를 사용하는 경우의 메시지 패싱

3.2 SPMD 방식의 구현 예

비동기 반복 알고리즘은 다양한 방법으로 구현될 수 있을 것이나, 본 논문에서 제시될 MPMD 방식과 비교하기 위해 [1]의 방법을 예로 들기로 한다.

동기 알고리즘에서는 연산이 동기화되므로 각 프로세스는 언제 어느 프로세스로부터 데이터가 올 지 예측이 가능하므로 그림 1과 같은 간단한 모델을 사용할 수 있다. 그러나 비동기 알고리즘에서는 그 예측이 불가능하므로 한편으로는 데이터를 기다리다가 즉시 처리함으로써 송신 프로세스의 대기 시간을 줄여야 하며, 동시에 자신이 담당한 계산을 수행해야 한다. [1]에서 사용한 메시지 패싱 모델은 그림 2와 같으며 모든 프로세스가 동일한 프로그램을 수행하는 SPMD(single program multiple data) 방식으로 간단히 구현된다(이하 타원은 프로세스, 직사각형은 CPU를 나타내기로 한다).

(그림 2) SPMD 방식의 대칭적 구현

인접 프로세스로부터 도착한 메시지가 있는지는 Algorithm 3과 같이 MPI_Iprobe()를 호출하여 flag의 값으로 알 수 있으며, 만일 도착한 메시지가 있다면 status라는 구조체를 사용하여 송신자(sender)와 보내온 데이터의 종류(msgkind)를 파악할 수 있다.

Algorithm 3. (for probing & receiving any arrived messages)

```
MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,
communicator,&flag,&status) ;
If (flag) {          /* flag is TRUE if there is a
message arrived. */
   sender = status.MPI_SOURCE ;
   msgkind = status.MPI_TAG;
   MPI_Recv( ... );   /* sender로부터 msgkind
   tag을 사용하여 message를 수신한다 */
   if (인접 프로세스가 경계치를 보냈다면)
      그 값을 저장하고 그 프로세스가 필요로
      하는 경계치를 보낸다;
   else  /* 인접 프로세스가 부분 잔차를 보냈
            다면 */
      그 값을 저장한다;
}
```

한편 각 프로세스가 수행하는 주연산은 Algorithm 4와 같다. 이는 Algorithm 2와 거의 같으나 데이터 수신 부분이 Algorithm 3으로 분리되어 있으며, 정지 요건으로는 인접 프로세스가 계산한 잔차의 값만을 참조한다는 점이 다르다 [1].

Algorithm 4 (for k-th process)

1. Algorithm 2와 동일
2. 만족스러울 때까지 다음을 반복한다.
   1) $\overline{x_{k_1}}, \ldots, \overline{x_{k_m}}$을 하나씩 순서대로 업데이트한다.
   2) 경계치 $\overline{x_{k_1}}$과 $\overline{x_{k_m}}$을 각기 좌측 및 우측의 인접 프로세스에게 보낸다.
   3) 부분 잔차 크기의 제곱 $\sum_{j=1}^{m} \| \overline{r_{k_j}} \|_2^2$를 계산하여 인접 프로세스에게 보낸다.
   4) 그간 수신한 부분잔차 정보를 이용하여

계산 계속 여부를 결정한다.

각 프로세스는 Algorithm 4를 수행하는 매 단계마다 Algorithm 3을 호출(심지어 시간이 가장 많이 걸리는 단계 2.1에서는 하나의 부분벡터를 계산할 때마다 호출할 수도 있다)하여 인접 프로세스의 대기시간을 줄이도록 한다. 그러나 Algorithm 3을 자주 호출하면 인접 프로세스의 대기시간은 줄일 수 있으나 정작 자신이 담당한 계산의 수행은 느려지게 된다. 또한 프로그램 곳곳에 Algorithm 3를 호출하는 부분을 넣어야 하므로 프로그램이 복잡하며, 얼마나 자주 호출해야 하는지에 따라 성능이 달라진다. 부하 불균형 상태에서 이 방법은 동기적 알고리즘보다 나은 결과를 주었다 [1].

4. MPMD 방식의 비대칭 송수신의 구현

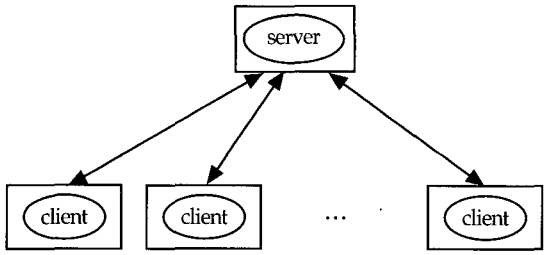
4.1 하나의 프로세스가 모든 기능을 수행할 경우의 문제점

MPI를 사용한 비대칭적 송수신의 구현에 있어 [1]은 §3.2의 방법을 사용하였으나 다른 방법으로 구현이 가능할 수도 있다. 그러나 근본적으로 노드당 하나의 프로세스만을 생성하여 비동기 알고리즘을 수행하는 경우, 그 프로세스는 CPU 시간을 나누어 두 가지 기능(즉 본연의 연산 기능 및 다른 프로세스의 대기 시간을 줄이도록 즉시 메시지를 수신하는 기능)을 겸할 수밖에 없으며, 한 기능의 향상은 다른 기능의 저하를 가져온다. 이는 비단 §3.2 의 방법만의 문제가 아니라 하나의 프로세스가 모든 일을 수행토록 하는 한 피할 수 없다.

따라서 연산과 메시지 수신 부분을 분리하여 각기 다른 프로세스가 담당하게 할 필요가 있으며, 이는 복수 개의 프로그램을 사용하는 MPMD(multiple program multiple data) 방식으로 간단히 구현할 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].

4.2 별도의 CPU에 서버를 두는 방법

프로세스들간의 비대칭적 전송을 구현하는 한 가지 방법은, 그림 3과 같이 별도의 서버프로세스를 두어 다른 프로세스들(이하 클라이언트라 부른다)은 메시지를 서버에게만 보내고, 필요한 데이터는 서버로부터 받도록 한다(서버는 공유 메모리의 역할을 한다). 그러면 송신자→서버, 서버→수신자 사이에는 송수신이 1:1이 되나, 최초 송신자와 최종 수신자 사이에는 송수신의 회수가 비대칭이 되며, 클라이언트들은 계산에만 전념할 수 있다.



(그림 3) 별도의 서버를 두는 경우의 메시지 패싱

서버 프로세스의 루틴은 다음과 같으며, MPI_Iprobe() 대신 단순히 MPI_Recv()를 사용하여 메시지가 올 때까지 대기하다가 수신한다.

Algorithm 5. (for server process)

```
do {
  MPI_Recv(buffer,count,MPI_Datatype,MPI_ANY
  _SOURCE,MPI_ANY_TAG,MPI_COMM_WOR
  LD,status)
  sender = status.MPI_SOURCE ;
  msgkind = status.MPI_TAG ;
  if (client가 경계치를 보냈다면)
      그 값을 저장하고 그 client가 필요로 하
```

는 경계치를 보낸다;
else /* client가 부분 잔차를 보냈다면 */
이를 사용하여 전체 잔차를 추정하고 작업의 계속 여부를 판단하고 알려준다;
} while (running) ;

running은 현재 작업중인 클라이언트의 수를 가리키며, 서버는 각 클라이언트가 보고한 부분 잔차를 사용하여 전체 잔차를 추정하고 만족스러우면 STOP 신호를 보내고 이 값을 1 감소시킨다. 모든 클라이언트가 종료하게 되면 running이 0이 되어 서버도 종료한다.

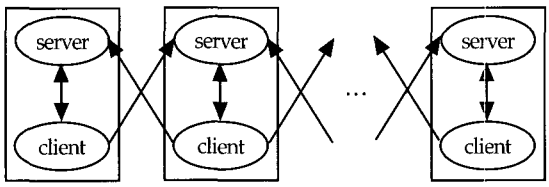
클라이언트의 알고리즘은 다음과 같다.

Algorithm 6 (for k-th client process)

1. 부분 벡터 $\overline{x_{k_1-1}}, \overline{x_{k_1}}, \ldots, \overline{x_{k_m}}, \overline{x_{k_m+1}}$의 초기치를 적절하게 택하고 대각 블록들 $A_{k_1k_1}, \ldots, A_{k_mk_m}$을 LU 분해한다.
2. 매스터 프로세스로부터 STOP 신호가 올 때까지 다음을 반복한다.
   1) $\overline{x_{k_1}}, \ldots, \overline{x_{k_m}}$을 하나씩 순서대로 업데이트 한다.
   2) 경계치 $\overline{x_{k_1}}$과 $\overline{x_{k_m}}$을 매스터 프로세스에게 보낸다.
   3) $\overline{x_{k_1-1}}$과 $\overline{x_{k_m+1}}$을 매스터 프로세스로부터 받는다.
   4) 부분 잔차 크기의 제곱 $\sum_{j=1}^{m} \| \overline{r_{k_j}} \|_2^2$를 계산하여 매스터 프로세스에게 보낸다.
   5) 매스터 프로세스로부터 계산의 계속 여부에 대한 명령을 수신한다.

4.3 CPU당 클라이언트와 서버를 하나씩 생성하는 경우

§4.1처럼 별도의 CPU에 서버를 두는 경우, 서버는 할 일이 너무 적어 하나의 CPU가 낭비되는 단점이 있다. 따라서 CPU당 클라이언트와 서버 프로세스를 하나씩 생성하고, 그림 4와 같이 클라이언트끼리 직접 송수신하지 않고 인접한 CPU 상의 서버에게 전달하며, 자신이 필요로 하는 데이터는 같은 CPU상의 서버에게 요청해 받도록 한다. 매스터 서버는 서버 중 하나를 택해 전체적인 수행을 관리하고 계산의 중지 여부를 결정하도록 한다.



(그림 4) 각 CPU에 서버와 클라이언트를 하나씩 생성하는 경우의 메시지 패싱

서버 프로세스의 알고리즘은 다음과 같다. [1]처럼 MPI_Iprobe() 대신 MPI_Recv()를 사용하여 매우 적은 CPU 시간을 소모하므로 같은 CPU상의 클라이언트의 수행에 지장을 주지 않는다. 매스터 서버는 개개의 클라이언트들로부터 직접 잔차의 크기를 수신하고 그것을 바탕으로 계산의 계속 여부를 결정하여 클라이언트에게 알려주도록 한다(그림에서 이 기능을 위한 메시지 패싱은 생략되어 있음).

Algorithm 7 (for k-th server process)

```
do {
  MPI_Recv(buffer,count,MPI_Datatype,MPI_ANY
  _SOURCE,MPI_ANY_TAG,MPI_COMM_WOR
  LD,status)
  sender = status.MPI_SOURCE ;
  msgkind = status.MPI_TAG ;
  if ((I am a master server) && (client가 부분
    잔차 크기를 보내오면))
      전체 잔차를 추정하고 작업의 계속 여부
      를 판단하고 알려준다;
   if (다른 CPU 상의 client가 경계치를 보냈다면)
```

그 값을 저장한다;
else if (같은 CPU 상의 client가 경계치를 요구하면)
요구하는 경계치를 전송한다;
} while (running) ;

Algorithm 8은 클라이언트의 알고리즘으로, 각자 계산한 경계치는 인접 서버에게 보내며, 자신이 필요로하는 경계치는 같은 CPU 상의 서버에게 요청한다는 점만 다르다..

Algorithm 8 (for k-th client process)

1. 부분 벡터 $\overline{x_{k_1-1}}, \overline{x_{k_1}}, \ldots, \overline{x_{k_m}}, \overline{x_{k_m+1}}$의 초기치를 적절하게 택하고 대각 블록들 $A_{k_1k_1}, \ldots, A_{k_mk_m}$을 LU 분해한다.
2. 다음을 반복한다.
   1) $\overline{x_{k_1}}, \ldots, \overline{x_{k_m}}$을 하나씩 순서대로 업데이트 한다.
   2) 경계치 $\overline{x_{k_1}}$과 $\overline{x_{k_m}}$을 각기 $k-1$ 및 $k+1$ 번째 CPU상의 서버에게 보낸다.
   3) $\overline{x_{k_1-1}}$과 $\overline{x_{k_m+1}}$을 같은 CPU 상의 서버에게 요청하여 받는다.
   4) 부분 잔차 크기의 제곱 $\sum_{j=1}^{m} \| \overline{r_{k_j}} \|_2^2$를 계산하여 매스터 서버에게 보낸다.
   5) 매스터 서버로부터 수신한 신호가 STOP 이면 계산을 끝낸다.

이 방법을 LAM/MPI 환경에서 시행하려면. 우선 server()와 client()를 완전히 분리된 독립 프로그램으로 만들고 컴파일한다. 모든 CPU에 클라이언트와 서버를 하나씩 생성하려면 그림 5와 같은 스키마 파일을 사용하여 % mpirun -v schema 명령으로 실행시킨다[10].

둘째 줄은 N(모든 노드)에 server() 프로세스를 생성하되 시행 모듈을 h(현재 로그인되어 있는 노드)로부터 가져다가 시행하라는 뜻이다. 셋째 줄은 모든 노드에 client() 프로세스를 띄우라는 것이다.

```
# mpirun schema file "schema"
server N -s h
client N -s h
```

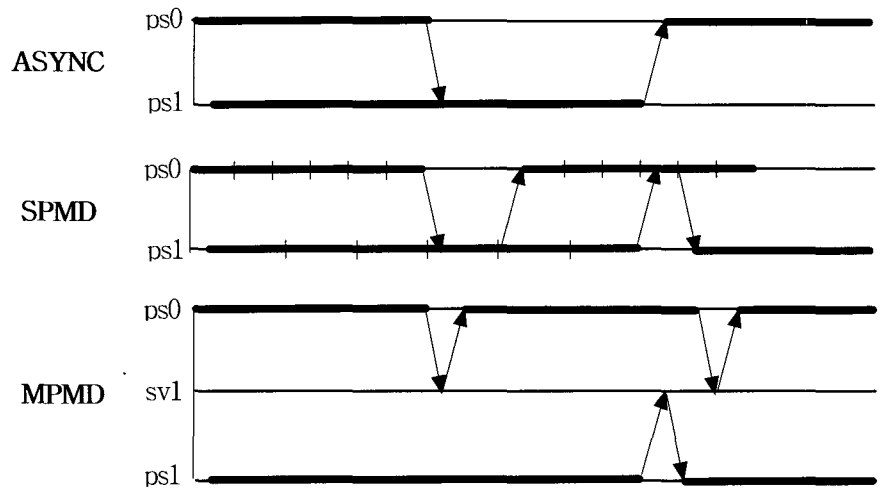
(그림 5) MPMD 방식 실행을 위한 schema 파일

5. 성능 실험 및 고찰

박과 신 [1]은, 이질 클러스터인 Atom 클러스터에서 본 논문과 동일한 문제를 사용하여 §3.2의 SPMD 방식의 수행 시간이 동기 병렬 알고리즘에 비해 17.5% 적게 걸림을 보였다. 따라서 본 실험에서는 이미 성능이 검증된 §3.2의 SPMD 방식과 새로이 제안한 §4.3의 MPMD 방식을 비교하도록 한다.

수원대학교의 Hydra 클러스터는 1대의 매스터 노드와 11대의 슬레이브 노드로 구성된 균질의 HPC 클러스터로서, 3Com 4900 series Gigabit 스위치로 상호 연결되어 노드 사이의 메시지 패싱에 걸리는 시간은 모두 동일하다. 실험에는 10개의 노드를 사용하여 행렬의 0이 아닌 성분만 나누어 저장하여 수행하도록 하였으며, 길이가 1,600만인 벡터들은 길이가 4,000인 부분 벡터 4,000개로 나누어 각 클라이언트로 하여금 400개씩 계산하도록 하였다. 한편 이질 클러스터를 시뮬레이션하기 위해, 특정 CPU에 계산 결과에는 영향을 미치지 않고 단순히 시간만 소모하는 추가적인 벡터 계산을 부과하였다.

표 1은 추가로 부과한 벡터 연산량 및 부과 노드의 위치에 따라, §3.2의 SPMD 및 §4.3의 MPMD 방식이 잔차의 크기가 2×10^{-9}에 도달할 때까지 걸린 실제 시간을 나타낸다. 그런데 각 CPU당 하나의 서버 프로세스를 항상 대기시켜 클라이언트의 요구를 즉각 처리함으로써 유휴 시간을 더욱 줄이는 동시에, SPMD 방식의 낭비적인 Algorithm 3의 호출을 제거하여 MPMD 방식이 더 나은 성능을 보일 것이라는 기대와는 달리, MPMD 방식은 SPMD



(그림 6) 여러 방식의 유휴 시간의 비교

(표 1) 잔차의 크기가 2×10^{-9}에 도달할 때까지 걸린 실제 시간

추가 부여 계산량		wall clock time(sec)	
부과 노드	추가 벡터연산	SPMD	MPMD
-	0	43.56	42.81
0	100	48.02	48.86
3	100	48.72	49.06
5	100	48.79	48.95
7	100	48.59	48.96
9	100	49.88	51.97
0	200	53.46	53.51
5	200	53.48	53.50
9	200	54.83	56.08
5	400	62.88	62.67
5	600	70.24	71.23
5	800	78.99	80.97
5	1,200	95.39	96.71
5	1,600	110.01	110.45
5	2,000	125.92	126.83

방식과 수행 시간이 거의 비슷한 결과를 보인다.

§3.2의 SPMD 방식은 도착한 메시지가 있는지 수시로 탐색해야 하며, MPMD 방식에서 클라이언트는 직접 다른 클라이언트로부터 데이터를 받지 않고 같은 CPU 상의 서버에게 요청하여(따라서 1회의 추가적 메시지 패싱을 필요로 한다) 받는다는 점이 각기 다른 방식에 비해 시간이 더 걸리는 부분이다. 그러므로 표 1의 결과로 볼 때, MPMD에서 서버가 메시지를 대기하였다가 수신함으로써 절약한 시간과 1회의 추가적 메시지 패싱에 걸린 시간이 거의 같음을 짐작할 수 있으며, 이는 메시지 패싱이 함수 호출보다 1단위 이상 시간을 더 소모한다는 사실도 그 이유중의 하나이다[11]. 또한 잘 알려진 바와 같이, Gigabit의 latency가 유난히 크다는 점도 한 원인으로 생각되며, 만일 Myrinet을 사용한다면 훨씬 좋은 결과를 얻을 것이라고 생각된다.

한편 같은 양의 추가 계산을 부과할 경우(추가 벡터 수가 100 및 200인 경우), 10개의 노드 0-9 중 대체로 마지막 노드에 부과할 때 약간 시간이 더 걸림을 볼 수 있는데, 이는 $\bar{x}$의 해 중 마지막 노드가 담당하는 부분 벡터의 성분의 크기가 크기 때문인 것으로 생각된다. 또한 노드 5에 추가적 부과 벡터의 수를 100개부터 2,000개까지 점차 증가시키며 그 노드의 성능을 다른 노드보다 떨어뜨려 계산한 결과로 볼 때, 두 방식 모두 노드 5의 성능 저하에 의한 유휴 시간의 증가를 억제하는 데 효과가 있음을 알 수 있다.

그림 6은 메시지 도착 여부를 수시로 탐색하지

않는 비동기 알고리즘(ASYNC), §3.2의 SPMD 및 §4.3의 MPMD 방식을 2개의 CPU를 사용하여 계산할 경우 연산 도중의 어느 시간 간격동안 실행시간과 유휴 시간을 나타낸 것이다. ps0 및 ps1은 각기 주 연산을 수행하는 빠른/느린 프로세스이며, MPMD의 sv1은 ps1과 같은 CPU 상에 존재하는 서버 프로세스를 나타낸다. 굵은 선은 프로세스가 연산에 소모하는 시간을 나타내며 가는 선은 유휴시간을 나타낸다. SPMD의 굵은 선 위의 짧은 세로 줄은 연산 도중 수시로 메시지의 도착 여부를 탐색하는 시점을 나타낸다. 화살표들은 프로세스 간의 메시지 패싱을 나타내며, SPMD 방식의 메시지 탐색 및 처리에 걸리는 시간과 MPMD의 서버가 메시지 처리에 사용하는 시간은 고려하지 않았다. 이를 보면, SPMD는 연산 도중 수시로 메시지를 탐색하고 처리함으로써 상대 프로세스의 대기 시간을 ASYNC에 비해 줄일 수 있고, MPMD는 독립된 서버 프로세스가 즉각 반응함으로써 대기시간을 더 줄일 수 있을 것이라고 예측할 수 있다.

실제 MPMD 프로그램이 시행되는 동안 Linux의 top 명령으로 확인한 결과 클라이언트 프로세스가 대부분의 메모리와 CPU 시간의 99.2% 이상을 점유하는 반면, 서버 프로세스는 적은 메모리와 0.5% 이하의 CPU 시간을 점유함이 관찰되었다.

향후 GRID와 같은 분산환경에서 수행할 경우, 특정 노드의 부하가 동적으로 변한다면 매 반복마다 일정한 수의 메시지 탐색을 수행하는 SPMD 방식보다 MPMD 방식은 적응형 방식으로서 더 나은 결과를 줄 것이라고 생각된다. 결론적으로 MPMD 방식은, 하나의 프로세스가 모든 역할을 수행하는 SPMD 방식의 여러 가지 문제, 즉 얼마나 자주 메시지를 탐색해야 최적의 결과를 얻을지 알 수 없고, 또한 프로그램이 복잡하고 수정하기 어렵다는 등의 단점을 보완하는 동시에, 클라이언트와 서버의 기능을 독립적으로 업그레이드 가능하다는 등 장점이 많으므로 나름대로 가치가 있다고 하겠다.

6. 결론 및 향후 연구

본 연구에서는 장차 GRID와 같은 지리적으로 분산된 계산 자원을 하나의 거대한 수퍼컴퓨터처럼 사용할 경우 문제가 될 노드들의 성능 차이 및 네트워크에 의한 전송 지연을 완화할 수 있는 하나의 방안으로 SPMD 및 MPMD 방식으로 비동기 반복 알고리즘을 구현하고 이를 크기가 1,600만×1,600만의 거대 희소행렬 문제에 적용하였다. 성능 실험에서 새로운 MPMD 방식은 이전의 SPMD 방식과 비슷한 성능을 보이나, SPMD 방식의 여러 가지 문제점을 보완하며 프로그램의 유지 관리가 쉽고 노드의 부하가 동적으로 변하는 경우 더 잘 적응할 수 있다는 장점이 있다.

향후 latency가 작은 고속 네트워크 상에서 성능 실험한다면 더 나은 결과가 얻어질 것으로 생각되며, 현재는 같은 CPU 상의 클라이언트와 서버 사이의 통신도 메시지 패싱을 통해서 이루어지므로 시간이 많이 걸리는데, 이를 공유 메모리를 통하여 데이터를 주고받는 방식이 개발된다면 MPMD의 성능이 더 나아질 것으로 생각된다. 한편, 본 연구는 각 노드의 부하가 동적으로 변하는 GRID와 같은 분산 환경을 사용한 연산에 적용하려면 추가적인 연구가 필요하다고 하겠다.

참 고 문 헌

[1] 박필성, 신순철, "비동기 알고리즘을 이용한 분산 메모리 시스템에서의 초대형 선형 시스템 해법의 성능 향상", 한국정보처리학회 논문지 8-A(4):439-446, 2001.

[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] 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, November 15~91, 1997.

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

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

[7] I. T. Foster, "Designing and building parallel programs," Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

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

[9] V. Migallón,, J. Penadés, and D. B. Szyld, "Nonstationary multisplittings with general weighting matrices", SIAM J. Matrix Analysis & Applications, Vol. 22, pp. 1089~1094, 2001.

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

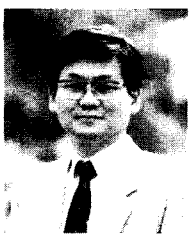
[11] P. S. Pacheco, "Parallel programming with MPI". Morgan Kaufmann Publishers, Inc., San Francisco, California, U.S.A., 1997.

[12] Y. Su and A. Bhaya, "Convergence of pseudo-contractions and applications to two-stage and asynchronous multisplitting for singular M-matrices", SIAM J. Matrix Analysis & Applications, Vol. 22, pp. 948~964, 2001.

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

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

◐ 저 자 소 개 ◑



박 필 성

1977년 서울대학교 해양학과 졸업(학사)
1984년 미국 올드도미니언대학교 대학원 계산 및 응용수학과 졸업(석사)
1991년 미국 메릴랜드대학교 대학원 응용수학과 졸업(박사)
1995년~현재 수원대학교 컴퓨터학과 부교수
1978년~1982년 KIST 해양연구소 연구원
1991년~1995년 한국해양연구원 선임연구원(전산실장, 수치모델그룹장 역임)
관심분야 : 고성능 컴퓨팅, 수치해석, 클러스터 컴퓨팅, GRID etc.
E-mail : pspark@suwon.ac.kr