표준 MPI 환경에서의 무정지형 선형 시스템 해법

A Fault-Tolerant Linear System Solver in a Standard MPI Environment

박 필 성*
Pil Seong Park

요 약

대규모 병렬 연산에 있어서, 계산 노드 혹은 통신 네트워크의 장애는 연산 실패로 끝나 계산자원이 낭비된다. 이를 해결하는 무정지형 MPI 라이브러리들이 제안되어 있으나 이들은 MPI 표준을 따르지 않아 이식성의 문제가 있다. 본 논문에서는 응용 프로그램의 수준에서 비동기 연산과 표준 MPI 함수만 사용하여 이식성의 문제를 해결하고 장애 복구 메커니즘을 단순화하며 수렴속도를 높이는 무정지형 선형 시스템의 해법을 제안한다.

Abstract

In a large scale parallel computation, failures of some nodes or communication links end up with waste of computing resources. Several fault-tolerant MPI libraries have been proposed so far, but the programs written by using such libraries have a portability problem since fault-tolerant features are not supported by the MPI standard yet. In this paper, we propose an application-level fault-tolerant linear system solver that uses the asynchronous iteration algorithm and the standard MPI functions only, which does not have a portability problem and is more efficient by adopting a simplified recovery mechanism.

☞ Keyword : 무정지(fault-tolerant), 선형시스템(linear system), MPI(Message Passing Interface), 비동기 반복 알고리즘 (asynchronous iteration algorithm)

1. 서 론

거대한 계산을 요하는 응용 프로그램들은 다수의 CPU를 가진 병렬시스템 혹은 GRID 환경에서 수행되는데, 이런 환경에서 문제가 되는 것은 연산 도중 일부 노드 혹은 통신회선 장애로 인한 연산실패이다. 따라서 사용자 입장에서는 무정지(fault tolerant) 해법이 중요한 이슈이다.

메시지 패싱(message passing) 방식은 프로세서간에 교환할 데이터를 메시지 전달함수를 사용해 주고받는 연산모델로서, 이런 함수들의 집합체인 메시지 패싱 라이브러리(message passing library)의 표준을 정한 것이 MPI(Message Passing Interface)[1]인데, 이에 맞추어 여러 MPI 라이브러리가 개발되어 있다. 그러나 MPI 표준은 아직 무정지 연산을 위한 해결책을 제시하지 않고 있어 여러 대학과 연구소에서 나름대로 무정지형 MPI 라이브러리를 개발하고 있다. 이들은 Co-Check MPI, MPI-CH V, LA-MPI처럼 체크포인트/롤백(checkpoint /roll-back)을 이용한 것과 MPI/FT 및 MPI-FT처럼 메시지나 프로세스의 복제(replication)를 사용하는 두 종류로 분류된다[2]. 그러나 이들은 일반적인 무정지 연산을 지원하기 위해 고안되어 상당한 오버헤드를 필요로 하며, 이식성이 떨어지고 대다수의 시스템은 이런 라이브러리를 가지지 않아 사용자 입장에서는 MPI 표준만을 사용한 무정지 연산기능을 응용 프로그램 수준에서 구현하는 것이 중요하다.

본 논문에서는 초대형 선형시스템 문제 해법에 있어서, 연산 도중 어느 계산 노드 또는 링크에 장애가 발생하더라도 전체적인 연산이 지

* 정 회 원 : 수원대학교 컴퓨터학과 부교수
pspark@suwon.ac.kr(제1저자)
[2005/06/14 투고 - 2005/06/20 1차 심사 - 2005/10/7 2차 심사 - 2005/11/09 심사완료]

속될 수 있는 무정지 해법을 제안하였다. 그러나 무정지형 MPI 라이브러리를 사용하지 않고, MPI 표준 함수만을 사용하여 응용 프로그램 수준에서 구현하여 이식성의 문제를 해결하였다. 또한 비동기 연산 방식을 도입하여 체크포인트/롤백이나 메시지 복제 기법 등을 사용하지 않고도 연산 도중 계산 노드간에 주고받는 데이터만을 사용하여 복구하도록 함으로써 복구 메커니즘을 단순화하고 오버헤드를 대폭 줄였으며, 부수적으로 더 나은 수렴 결과를 얻을 수 있도록 하였다.

2. 관련 연구 및 대상 문제

2.1 비동기 반복 알고리즘

일반적인 병렬연산의 경우, 직렬연산과 같은 결과를 얻기 위해 여러 단계에 동기점을 설정하여 모든 계산 노드(이하 프로세서라 부른다)를 동기화하는 동기 알고리즘(synchronous algorithm)의 사용이 일반적이다. 그러나 프로세서의 성능이 서로 다르거나 부하균형이 어려운 경우, 성능은 가장 느린 프로세서에 의해 결정된다.

이런 문제를 극복하기 위해 알고리즘 내의 동기점을 없애고 각 프로세서가 비동기적으로 유휴시간 없이 수행하도록 하여 전체적인 성능을 높이고자 고안된 것이 비동기 반복(asynchronous iteration) 알고리즘인데, 이는 본 논문에서 핵심적 역할을 한다. 비동기 반복 알고리즘은 Chazan & Miranker[3]에 의해 처음 제안된 이래 그 기법은 수치적인 문제를 다루는 데 있어서 많은 발전이 이루어졌으며, 일반적인 전산학 문제 등 점차 사용 영역이 확대되고 있다[4,5]. 본 논문은 초대형 희소 선형시스템(large sparse linear system) $A\overline{x} = \overline{b}$(A는 $n \times n$ 행렬, $\overline{b}$는 길이 n의 벡터)의 해법을 다루므로, 비동기 반복 알고리즘도 선형시스템 문제에 국한하기로 한다.

수학적으로 $A\overline{x} = \overline{b}$는 (1)처럼 $L \times L$ 블록 형태로 쓸 수 있다.(A_{ij}, $i, j = 1, \cdots, L$은 A의 부분행렬, $\overline{x_1}, \overline{x_2}, \cdots, \overline{x_L}$과 $\overline{b_1}, \overline{b_2}, \cdots, \overline{b_L}$은 각기 $\overline{x}$와 $\overline{b}$의 부분 벡터들이다.)

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

이는 L개의 부분문제들 (2)의 조합과 동일하다.

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

k번째 부분문제는 LU 분해 등을 사용하여 $\overline{x_k}$를 계산할 수 있다. 그러나 (2)의 우변은 다른 부분문제의 해벡터 $\overline{x_j}(j \neq k)$를 포함하므로 직렬연산에서는 수렴할 때까지 부분문제들을 순서대로 반복해 풀어야 한다. 동기 병렬 알고리즘에서는 프로세서당 하나씩 부분문제를 할당하여 독립적으로 구한 부분 해벡터를 동기점에서 서로 교환하고 우변을 업데이트한 후 다시 $\overline{x_k}$를 구하는 과정을 반복한다.

비동기 알고리즘의 경우는 동기점이 없으므로, 각 프로세서는 다른 프로세서와 무관하게 계산을 진행하고 연산과 메시지 패싱은 비동기적으로 이루어진다. 공유 메모리 시스템에서는, 단순히 각 프로세서가 자신의 작업진도에 따라 업데이트한 부분 해벡터를 공유 메모리에 저장하고 필요할 때마다 다른 부분 해벡터를 읽도록 함으로써 쉽게 구현할 수 있다[6].

Algorithm 1 (공유 메모리 시스템에서의 k번째 프로세서의 비동기 알고리즘)

1. 초기치 부분벡터들 $\overline{x_j}(j \neq k)$를 적절히 선택한다.
2. 다른 프로세서와 무관하게 각자 다음을 반복한다.
   1) $A_{kk}\overline{x_k} = \overline{b_k} - \sum_{j \neq k} A_{kj}\overline{x_j}$를 풀어 $\overline{x_k}$를 갱신하고 공유 메모리에 저장한다.
   2) 다른 프로세서가 계산한 $\overline{x_j}(j \neq k)$를 공유 메모리로부터 읽는다.
   3) 부분 잔차 $\overline{r_k}$를 계산하고 이를 공유 메모리에 저장한다.
   4) 대표 프로세서는 모든 프로세서가 계산하여 저장한 부분 잔차를 종합해 수렴 여부를 판단하고 연산의 계속 여부에 대한 지시를 공유 메모리 변수 (예를 들어 GoStop이라 하자)에 저장한다.
   다른 프로세서들은 변수 GoStop을 읽어 연산의 계속 여부를 판단한다.

k번째 프로세서의 부분잔차 $\overline{r_k}$는 다음과 같이 정의하며(즉 좌우변의 차이), 이들이 조합인 전체 잔차의 크기는 수렴의 척도로 사용된다.

$$\overline{r_k} = \overline{b_k} - [A_{k1} \ A_{k2} \cdots A_{kL}] \ \overline{x}$$

공유 메모리 시스템에서의 동기 알고리즘은, 2.2) 및 2.4) 직전에서 모든 프로세서가 그 단계에 이를 때까지 대기하는 동기화 과정이 필요하므로 빠른 프로세서는 대기해야 하며, 모든 프로세서가 같은 회수만큼 루프를 반복하게 된다.

그러나 비동기 알고리즘의 경우, 각 프로세서는 유휴시간이 없이 성능 및 부하에 따라 각기 다른 회수만큼 연산을 반복한다. 따라서 다른 프로세서가 갱신하는 $\overline{x_j}(j \neq k)$의 값은, 갱신 여부와 상관없이 공유 메모리에 저장되어 있는 그대로 읽어서 사용한다. 따라서 빠른 프로세서는 느린 프로세서가 $\overline{x_j}$를 갱신하기를 기다릴 필요가 없고, 느린 프로세서는 사용하려는 값이 빠른 프로세서에 의해 여러 번 갱신되더라도 가장 최근 값만을 계산에 이용하므로 프로그램 수행의 실제 시간(wall clock time)은 동기 알고리즘보다 1/3-1/2까지 단축된 결과도 보고되고 있다 [7,8].

비동기 알고리즘을 분산 메모리 환경에서 구현하는 것은 그리 간단하지 않다. 그 이유는, MPI에서는 어떤 프로세서가 메시지를 보내면 수신 프로세서가 그것을 수신하고 응답을 해야 송신 프로세서가 다음 계산을 진행할 수 있기 때문이다. 연산 프로세서들이 비동기적으로 작동하도록 하기 위하여 [9]에서는 서로간에 직접 연산결과를 교환하지 않고, 별도의 데이터 전달 프로세서를 통해 자신의 진도에 따라 하도록 함으로써 구현하였다.

비동기 연산 방식은 비결정적(nondeterministic) 결과를 주므로 옳은 해로의 수렴여부는 명백하지 않다. 그러나 비동기 반복 알고리즘의 수렴이 보장되기 위한 행렬의 조건은 잘 알려져 있으며 [6], 그 성질을 만족하는 한 얼마든지 비동기적으로 연산을 하더라도 궁극적으로 항상 옳은 결과가 보장된다. 그 조건은 특수한 성질이 아니라 상당수의 자연과학 계산에서 볼 수 있는 비교적 일반적인 성질로서 그 조건을 만족하는 문제를 다루기로 한다.

2.2 MPI(Message Passing Interface) 함수들

본 논문에서 사용하는 LAM MPI 환경에서는 계산 주체가 프로세서가 아니라 프로세스이며 사용자는 프로그램 실행시 필요한 프로세스의 개수를 지정하는데, 각 프로세스는 랭크(rank, 즉 번호)에 의해 구분된다.

MPI의 대표적인 1:1 메시지 송신 및 수신 함수는 다음과 같다.

int MPI_Send(void* buffer, int count, MPI_Datatype datatype, int destination,
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 데이터의 형, count는 데이터의 수, destination과 source는 각기 수신자 및 발신자의 랭크, tag은 메시지를 구분하는 번호, 그리고 status는 수신 결과와 관련된 정보를 저장한 구조체이다. 송신자에 의한 MPI_Send() 함수의 호출은 수신자의 MPI_Recv() 호출에 의해 처리되어야 하며, 이 경우 count, datatype, tag, communicator가 일치해야 하고, destination과 source는 상대방의 랭크를 사용한다.

한편 언제 어디서 올 지 모르는 메시지는, MPI_Iprobe() 함수를 사용하여 자신에게 온 메시지가 있는지를 확인하고, 만일 도착한 메시지가 있을 경우에는 status라는 구조체를 사용하여 발신자와 태그를 파악하여 MPI_Recv() 함수를 호출함으로써 수신할 수 있다.

MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status)

source와 tag은 특정 프로세스의 랭크 및 메시지 번호를 지정하는 것이 일반적이나, MPI_ANY_SORUCE 및 MPI_ANY_TAG처럼 지정하면 어떤 송신자가 어떤 택을 붙여 보내더라도 그 메시지를 수신할 수 있다.

2.3 대상 문제

대상 문제로는 Kaufman[10]에 기술된 2큐 오버플로 큐잉 모델(overflow queuing model)을 약간 변형한, 비동기 알고리즘의 수렴 요건을 만족하는 선형 시스템 $A\bar{x}=\bar{b}$을 다룬다. 행렬 A의 그래프는 3,000×3,000의 2차원 격자모양이며, 크기는 900만×900만인 거대 희소행렬로서 구하려는 벡터 $\bar{x}$의 각 성분은 격자점에서의 값이다.

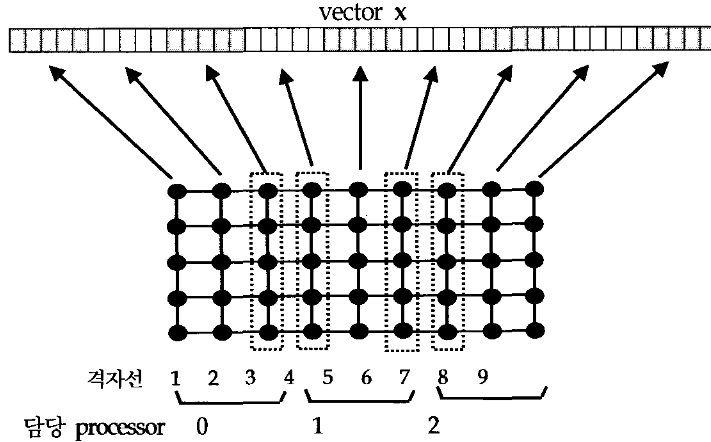
이 문제를 (2)와 같이 부분문제들로 나누어 여러 프로세서를 사용한다 하더라도 행렬 전체를 그대로 메모리에 저장하는 것은 불가능하므로(배정밀도 변수를 사용하면 행렬 A의 저장에만 최소 640TB($=(9\times10^6)^2\times8$ bytes)가 필요함), 0이 아닌 성분만 벡터 형태로 저장하여 계산을 하는 특수한 기법을 사용하였다.

3. 일반적인 동기 병렬 알고리즘

우선 분산 메모리 시스템에서의 일반적인 동기 병렬 알고리즘의 예를 보자. 대상문제 같은 2차원 격자문제는 1차원 영역분할(domain decomposition)을 사용하여 여러 개의 작은 문제로 나누어 블록 가우스-사이델법(block Gauss-Seidel)을 적용할 수 있다.

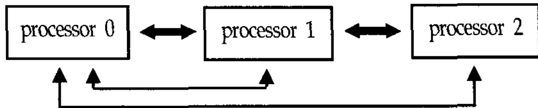
예를 들어, 그림 1처럼 9×5의 격자 문제를 3개의 프로세서로 계산하는 경우, $A\bar{x}=\bar{b}$의 행렬 A는 45×45이며 해 벡터 $\bar{x}$의 길이는 45가 된다. 9개의 수직 격자선을 프로세서 당 3개씩 계산하도록 작업을 분배한다. 각 격자점에서의 값을 계산하려면 인접한 격자점에서의 값들이 필요하다. 그런데 각 프로세서가 담당한 영역의 경계 격자선(점선으로 표시) 상의 값의 계산에는 다른 프로세서가 담당한 경계 격자선에서의 최신값이 필요하므로(예를 들어, 격자선 3의 계산에는 격자선 2와 4의 값이 필요한데, 격자선 4는 프로세서 1이 업데이트한다) 프로세서끼리 경계 격자선 상의 값을 교환해야 한다.

프로세서들은 개념적으로 그림 2와 같이 나열되어 있다고 생각할 수 있고, processor 0이 전



〈그림 1〉 9×5의 격자문제를 3개의 프로세서로 계산하는 경우의 영역분할 및 전체 벡터 $\overline{x}$에서의 부분 벡터들의 위치

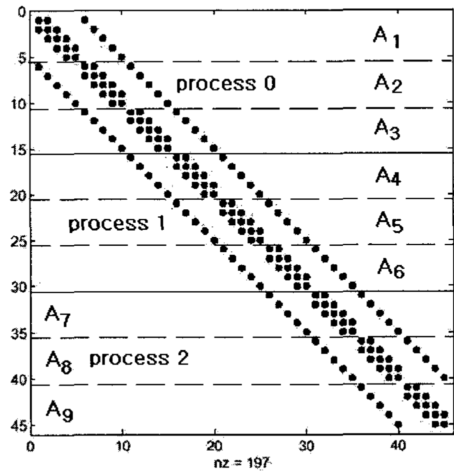
체적인 컨트롤을 담당하도록 하면 프로세서들 간의 데이터 교환은 굵은 화살표로 표시된 경로로 이루어지며, 컨트롤은 가는 화살표를 통해 이루어진다.



〈그림 2〉 3개의 프로세서를 사용하는 경우의 메시지 패싱

위 문제의 행렬은 5대각 희소 행렬이며, 0이 아닌 성분들만을 점으로 표시하면 그림 3과 같다. 행렬 A를 5행씩 부분행렬로 나눈 것을 $A_1, \cdots, A_9$이라 하고, 각 프로세서는 3개씩의 부분행렬만 가지고 연산을 수행한다.

각 프로세서는 (2)처럼 담당한 영역의 모든 격자점의 값을 일시에 계산할 수도 있으나, 그림 1의 각 수직 격자선을 작업단위로 하여 연산을 한다면 작업 크기가 작아 효율적이며 이것이 일반적으로 사용되는 방법이다. 즉 각 프로세서는 수직격자선 3개를 하나씩 차례로 업데이트한 후 자신의 경계치(그림 1에서 점선 사각형 속의 격자점의 값)를 인접 프로세스에게 전달하고 또한 인접 프로세스가 업데이트한 경계치를 받아서



〈그림 3〉 축소된 문제의 행렬 구조와 부분 행렬로의 분할

다음의 연산을 진행한다. 따라서 다음과 같은 동기 병렬 알고리즘을 사용할 수 있는데, 3.2)와 3.7)에서 2회의 동기화가 필요하다.

Algorithm 2 (각 프로세서의 동기 알고리즘)

1. 자신의 프로세서 번호, 이웃 등을 파악한다.
2. 연산을 위해 자신의 부분 행렬 및 초기치 등을 셋업 한다.

3. processor 0으로부터 정지하라는 지시가 올 때까지 다음을 반복한다.
   1) 담당한 격자선상의 값을 순서대로 1회씩 업데이트한다.
   2) 모든 프로세서가 이 지점에 도달할 때까지 기다린다.
   3) 인접 프로세서와 업데이트한 경계치를 교환한다.
   4) 최신 정보를 이용하여 자신의 부분 잔차를 계산한다.
   5) 모든 프로세서는 계산된 부분 잔차의 크기를 processor 0에게 보낸다.
   6) processor 0은 이들 부분 잔차를 받아 자신의 것과 더해서 수렴 여부를 판단하고 다른 프로세서에게 계산의 계속 여부에 대한 지시를 내린다.
   7) 다른 프로세서들은 processor 0으로부터 계산의 계속 여부에 대한 지시를 받는다.

4. 비동기 연산을 이용한 무정지형 해법

4.1 무정지형 해법의 목적

분산 메모리 환경에서 거대 선형시스템 해법의 병렬 연산을 수행함에 있어 어느 프로세서나 링크의 장애는 연산 실패를 가져온다. 따라서 감시 프로세서가 이를 탐지하여 다른 프로세서로 하여금 그 역할을 대신하여 연산을 계속 수행토록 함으로써 중단없이 완료할 수 있는 시스템을 MPI 표준함수만을 사용하여 응용 프로그램 수준에서 구현하고자 한다.

4.2 비동기 연산의 필요성

MPI에서 1:1 송수신의 경우 장애가 발생한 프로세서와 통신하려던 다른 프로세서들은 무한정 대기하게 된다. 만일 동기 알고리즘을 사용한다면 동기점에서 모두 데이터를 교환하려 하므로 더 이상의 진행이 불가능하며, 표준 MPI 함수로는 이 상황을 벗어날 방법이 없다. 또한 무정지형 연산을 위해 동기 알고리즘을 사용한다면 다음과 같은 문제가 있다.

1) 모든 프로세서는 새로 계산한 부분 벡터를 체크포인팅을 위해 동시에 어딘가로 보내야 하므로 네트워크 상에 일시적으로 엄청난 트래픽이 유발되며 충돌 확률이 높다.
2) 장애가 복구될 때까지 모든 다른 프로세서들도 연산을 중단하고, 최근 체크포인팅된 시점으로 롤백해야 한다. 이는 정상적인 프로세서의 작업을 인터럽트 해야 하며 복구 메커니즘이 복잡하고 비효율적이다.

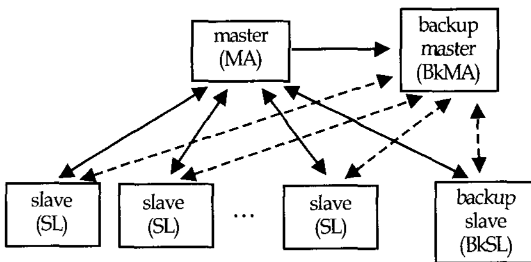
반면 별도의 데이터 교환 중계 프로세서를 사용한 비동기 알고리즘의 이점은

1) 모든 프로세서는 자신의 작업 진도에 따라 데이터 교환 중계 프로세서에게 데이터를 보내므로 데이터 전송이 시간적으로 분산된다.
2) 복구 프로세서는 데이터 교환 중계 프로세서로부터 장애가 발생한 프로세서의 최종 계산치를 받아서 연산을 시작하고, 다른 프로세서들은 복구 과정과 무관하게(즉 롤백 불필요) 하던 연산을 계속하면 된다. 따라서 복구 메커니즘이 간단하고 복구하는 동안에도 전체적인 연산은 수렴을 계속하게 된다.

4.3 무정지형 해법의 모델

따라서 비동기 알고리즘을 사용하며, 그림 4처럼 별도의 데이터 교환 중계 프로세서(매스터, 즉 **MA**로 표기)를 둔다. 연산 프로세서들(슬레이브, 즉 **SL**로 표기)은 업데이트한 자신의 경계

치를 MA에게 보내고, 자신이 필요로 하는 이웃 SL이 계산한 경계치는 MA로부터 받도록 한다 (실선 화살표). MA는 SL들이 보고한 모든 경계치를 저장하며 SL들간의 데이터 교환을 중계하고, 각 SL에게 장애가 발생하지 않는지 감시하도록 한다. 한편 어느 SL의 장애가 감지되면 백업 슬레이브(**BkSL**로 표기)를 기동하여 그 역할을 대신하도록 한다.



〈그림 4〉 무정지형 매스터-슬레이브-백업 모델

그러나 만일 MA가 어느 SL에게 데이터를 요청한 후 그 SL에게 장애가 발생한다면, MA는 무한정 응답을 기다리게 되며 전체 시스템이 마비될 것이다. 따라서 모든 데이터 교환은 SL의 요청에 의해 시작되도록 하고, MA는 단지 수동적으로 반응하도록 한다.

MA는 연산을 하지 않고 데이터 중계와 컨트롤에만 관여하므로 부하가 아주 적어 SL들에 비해 장애발생 확률이 아주 낮다. 그러나 모든 데이터가 MA로 집중되므로 데이터 양을 줄일 필요가 있다. 한편 MA에 장애가 발생하는 경우를 대비하여 백업 매스터(**BkMA**로 표기)를 두어 MA에 장애가 발생하면 모든 SL의 요구를 대신 처리하도록 한다(점선 화살표).

그림 4에서 각 프로세서가 취급하는 중요한 데이터는 다음과 같다.(n은 SL의 수)

1) SL 및 BkSL : 자신이 담당한 영역의 격자점에서의 값으로 이루어진 부분벡터(길이 900만/ n) 및 이의 연산에 사용되는 부분행렬(그림 3 참조. 이는 1차원 배열에 저장)
2) MA 및 BkMA : 각 SL이 담당한 영역의 최좌측 및 최우측 격자선 상의 값으로 이루어진 부분벡터들(길이 3,000인 벡터 2 n개)

중요한 것은, MA와 BkMA가 유지하는 데이터는 일반적인 병렬 연산에서도 필히 교환되어야 하는 것이며, 체크포인팅 등의 목적으로 별도의 데이터를 전송하거나 저장하지 않는다는 점이다. 이에 대해서는 §4.5에서 더 자세히 논의한다.

4.4 장애 탐지

MA는 어떤 SL(i번째라 하자)이 부분잔차의 크기를 보고하는 시각 $t_{last}(i)$ 및 평균 간격 $t_{avg}(i)$를 계산하여 기억한다. 만일 i번째 SL이 $t=t_{last}(i)$에 잔차를 보고하면, MA는 다른 j번째 SL($j\neq i$)들이 $t-t_{last}(j) < TOL \times t_{avg}(j)$ 조건(TOL은 적절히 지정할 허용계수)을 만족하는지를 확인하고 그렇지 않으면 장애를 일으킨 것으로 간주한다.

MA의 장애에 대비하여, 모든 SL(및 BkSL)과 MA는 서로 주고받는 모든 메시지를 BkMA에게도 동시에 보내도록 한다. 그러면 BkMA는 MA와 SL들간에 일어나는 모든 상황을 알 수 있으며(그러나 BkMA는 답을 하지 않는다), MA와 동일한 데이터를 유지할 수 있다. 만일 MA가 어떤 SL의 데이터 전송 및 요구를 받고도 지정된 시간이 지나도록 응답하지 않으면, BkMA는 MA에 장애가 발생한 것으로 간주하여 SL들에게 알리고 그 역할을 대신한다.

4.5 복구와 관련된 문제

어떤 SL에 장애가 발생하면 MA(혹은 BkMA)는 BkSL에게 장애 발생한 SL의 랭크와 그 SL

이 최근 보고한 좌우 경계치(길이 3,000인 벡터 2개)를 주어 역할을 대신하도록 한다. BkSL은 장애 발생 SL의 랭크만 알면 그 SL 역할의 수행에 필요한 부분행렬을 간단히 생성할 수 있다. 따라서 체크포인팅을 위해서 별도로 다른 데이터를 전송하거나 저장하지 않고 단순히 병렬 연산 단계에서 MA에 저장되는 데이터만으로 복구를 수행하도록 한다.

병렬연산에서 프로세서들간에 필히 주고받아야 하는 것은 각자 담당한 영역의 경계 격자선 상의 값이므로(그림 1 참조). 그림 4에서도 MA(혹은 BkMA)는 각 SL들이 계산한 경계치만 받아서 저장하고 다른 SL이 요구하면 전달한다. 그러나 BkSL이 장애 발생 SL의 역할을 떠맡기 위해서는 내부 격자선 상에서의 값도 필요한데, 그러면 결국 모든 각 SL이 계산하는 길이 900만의 배정밀도 벡터 전체($9\times10^6\times8$ bytes)를 체크포인팅 등의 방법으로 수시로 어딘가 저장해야 한다. 이것의 전송에는 Gigabit LAN에서 대략 1초 정도, 실제로는 지연시간만큼 더 걸리게 되는데, 이는 Algorithm 2의 1회 연산에 필요한 시간보다 훨씬 길다. 또한 이 데이터를 MA(혹은 BkMA)가 아닌 제3의 장소에 저장한다는 것은 낭비이다.

문제는, BkSL이 장애 발생 SL이 담당한 영역의 경계치만을 사용해서 그 역할을 대신할 수 있는가 하는 것이다. 다행히 대상문제와 같은 거대문제는 인접 격자점에서의 값은 서로 비슷하며 전체적으로 급격한 변화없이 값의 분포는 연속적이므로 경계치로부터 보간법을 사용하여 유사한 값을 만들어 낼 수 있다. 예를 들어, 그림 1의 경우, 만일 프로세서 1에 장애가 발생하면 BkSL은 격자선 4, 6상의 값을 사용하여 선형보간법으로 격자선 5의 값을 만들어 사용하도록 한다.

이를 위해서는 Algorithm 2와는 달리, 모든 SL들은 MA와 BkMA에게 경계치를 보낼 때 좌우 경계치를 모두 보내야 한다. 예를 들어, 그림 1에서 SL 0는 §3에서는 격자선 3에서의 값만 보고했으나, 격자선 1의 값도 보고하도록 수정해야 한다.

n개의 SL로 연산을 수행할 경우의 데이터 전송량을 계산해보자. 각 SL이 매 반복당 MA와 BkMA에게 각기 전송하는 것은 좌우 경계격자선 상의 길이 3,000의 부분벡터 2개뿐이다. 따라서 n개의 SL이 1회 전송하는 데이터의 총량은 $2\times2\,n\times3{,}000\times8$ bytes로서 $96\,n$ KB에 불과하다. 그러나 중간계산 결과 벡터 전체를 체크포인팅하며 병렬 연산한다면 연산을 위한 데이터 교환에 $2(n-1)\times3{,}000\times8$ bytes, 체크포인팅에 900만$\times8$ bytes의 전송이 필요할 것이다.

4.6 무정지형 비동기 알고리즘

슬레이브(SL)의 알고리즘은 Algorithm 2와 비슷한 구조이나, 비동기 알고리즘을 사용하고 MA와 BkMA에게 양쪽 격자선상에서의 경계치를 모두 전송한다는 점과, 자신이 필요로 하는 데이터는 MA(혹은 BkMA)에게 요청해 받는다. 또한 동기점이 없으므로 각자 자신의 부분잔차의 크기로 수렴에 대한 판단을 하여 정지 여부를 결정하며, MA와 BkMA는 3.5)에서 보내온 부분 잔차의 크기로 그 SL의 연산 종료 여부를 알 수 있다.

Algorithm 3 (SL 프로세서들)

1. 자신의 프로세서 번호, 이웃 등을 파악한다.
2. 연산을 위해 자신의 부분 행렬 및 초기치 등을 셋업한다.
3 충분히 수렴할 때까지 독립적으로 다음을 반복한다.
   1) 자신이 맡은 수직 격자선의 값을 하나씩 순서대로 업데이트한다.
   2) MA와 BkMA에게 자신의 양쪽 경계치를 보낸다.

3) MA와 BkMA에게 자신이 필요한 인접 SL이 계산한 경계치를 요청하고 MA(MA의 장애시는 BkMA)로부터 이를 수신한다.
4) 최신 정보를 이용해 자신의 부분 잔차를 계산한다.
5) 계산된 부분 잔차의 크기를 MA와 BkMA에게 보낸다.
6) 자신의 부분 잔차가 충분히 작으면 연산을 끝낸다.

백업 슬레이브(BkSL)는 프로그램 수행 시작 초기에 미리 띄워서 대기시킬 수도 있고 MPI_Spawn() 함수를 써서 장애 발생이 탐지되는 순간 기동시킬 수도 있다.

Algorithm 4 (BkSL 프로세서)

1. 자신의 프로세서 번호, 이웃 등을 파악한다.
2. MA(혹은 BkMA)로부터의 메시지를 기다린다.
   1) 만일 끝내라는 명령이 오면 시행을 종료한다.
   2) 만일 어느 SL의 역할을 대신하라는 명령이면 Algorithm 3의 2단계부터 시행하여 연산에 참여한다.

동기 알고리즘은 연산이 동기화되므로 MA(혹은 BkMA)는 어느 SL이 언제, 어떤 데이터를 보내올지 알 수 있으나 비동기 알고리즘은 그렇지 않다. 한 가지 해법은, Algorithm 5와 같이 MPI_Iprobe() 함수에 인자를 MPI_ANY_SOURCE와 MPI_ANY_TAG을 사용하여 도착한 메시지가 있는지 확인하고, status라는 구조체를 사용하여 송신자 status.MPI_SOURCE와 메시지 태그 status.MPI_TAG을 파악하여 MPI_Recv() 함수를 호출함으로써 보내온 메시지를 수신하고 요청에 따라 적절한 조치를 취하도록 한다.

Algorithm 5 (보내온 메시지를 탐지하고 수신하기)

```
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( ... );  /* Receive a message from sender */
  Perform appropriate jobs.
}
```

매스터(MA) 및 백업 매스터(BkMA)의 알고리즘은 각기 다음과 같다.

Algorithm 6 (MA 프로세서)

1. 모든 SL이 종료할 때까지 다음을 반복한다.
   1) Algorithm 5로 SL들로부터 온 메시지가 있는지 확인하고 메시지의 발신자(i번째 SL이라 하자)와 종류를 파악한다.
   2) 메시지의 종류에 따라 다음을 시행한다.
      a. 만일 i번째 SL이 경계치를 보냈다면, 이를 저장하고 그 SL이 필요로 하는 인접 SL의 경계치를 보낸다.
      b. 만일 i번째 SL이 부분 잔차를 보냈다면
      - 이를 저장하고
      - 모든 $j(j \neq i)$번째 SL에 대해, 그 SL이 최종적으로 잔차를 보낸 후 경과된 시간이 TOL* $t(j)$ 크면 BkSL을 기동한다.

Algorithm 7 (BkMA 프로세서)

1. 모든 SL이 종료할 때까지 다음을 반복한다.
   1) Algorithm 5로 SL들 혹은 MA로부터 온 메시지가 있는지 확인하고 발신자와 태그를 파악하여 이를 수신하고 저장하며, SL의 요청 이후 MA가 응답할 때까지의 시간 t_{resp}을 측정한다.
   2) 지금까지 측정된 t_{resp}의 평균 t_{avresp}를 구하고
      a. 만일 MA가 t_{avresp}의 k_{MA}배의 시간이 지나도록 SL에게 응답을 보내지 않으면 MA의 장애를 간주하고 각 SL에게 MA의 장애를 알리며 Algorithm 5를 수행하여 MA를 대신한다.
      b. 그렇지 않으면 GOTO 1.1).

5. 성능 평가

본 실험에서는, 어느 SL 혹은 MA에 장애가 발생할 경우 연산이 재개되는 데 걸리는 시간을 알아보고, 전체적인 수렴속도를 알아보고자 한다. 실험에는 수원대학교의 Hydra 클러스터를 사용하였는데, 이는 Xeon CPU를 가진 12개의 노드로 구성된 균질의 HPC 클러스터로서, 3Com 4900 series Gigabit 스위치로 상호 연결되어 있다. 실험에는 MA와 BkMA, BkSL 외에 4개의 SL을 사용하여 행렬의 0이 아닌 성분만 저장하여 수행하도록 하였다. MA의 장애는 일정 시간이 지나서 수행을 종료하도록 하여 시뮬레이션 하였고, SL의 장애는, 프로그램에서 그 SL의 랭크를 지정하고, Algorithm 3의 루프를 n_d회 반복을 시작할 때 프로그램의 수행을 종료하도록 함으로써 시뮬레이션 하였다.

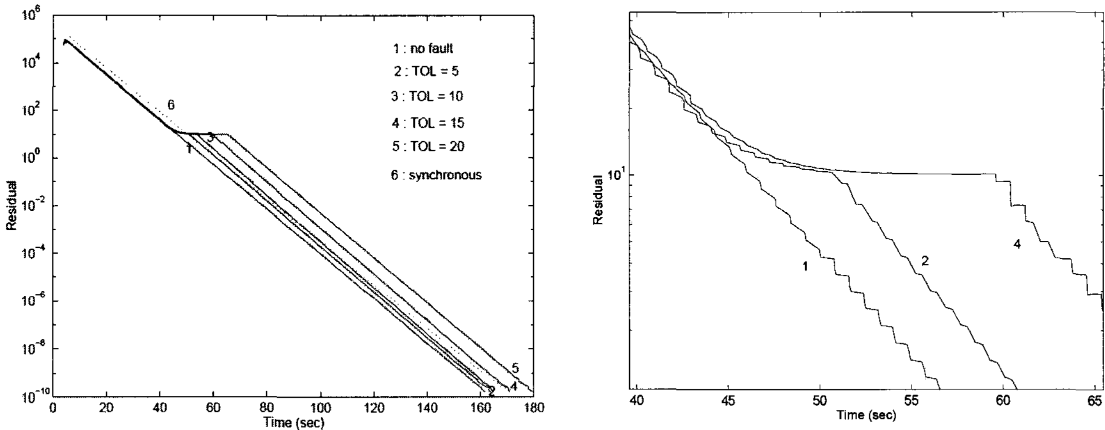
표 1은 다양한 n_d와 TOL 값에 대해 두 번째(즉 랭크 1) SL에게 장애가 발생한 후 MA가 이를 파악하여 BkSL을 기동하여 대신할 때까지 걸린 실제 시간을 나타낸 것이다.

〈표 1〉 장애 발생 후 백업 프로세서가 계산을 시작할 때까지 걸린 시간(초)

n_d	TOL			
	5	10	15	20
10	12.59	14.31	21.54	27.32
50	7.85	11.35	15.91	20.78
100	7.76	11.18	15.65	19.67

n_d가 50 및 100인 경우는 거의 같은 결과가 나왔으나 10일 경우는 그 값이 비교적 큰데, 이는 계산을 시작한지 얼마 지나지 않아 MA가 SL들의 t_{avg}값을 제대로 정확히 파악하지 못했기 때문이다. TOL의 값을 너무 작게 주면, SL에 장애가 발생하지 않아도 MA가 잘못 판단할 수 있겠으나, 이 결과로 볼 때, 계산이 어느 정도 진행된 이후 장애가 발생한다면 TOL의 값은 5보다 작은 값(실제 2 혹은 3도 괜찮다는 결론을 얻었다)이라도 무난할 것으로 사료된다. 추가적 계산에 의하면, BkSL이 MA에 의해 기동된 후 장애가 생긴 SL을 대신하는 데 걸린 시간은 대략 2.4-3.1초 사이인 것으로 추정된다. 한편 [2]는 크기가 2,677,324인 CG(conjugate gradient) 해법을 16개의 프로세서로 계산할 경우 69초의 복구 시간이 소요되었고 이는 전체 연산의 6%에 해당된다. 반면 문제가 달라 직접적인 비교는 불가능하나, 본 논문의 무정지형 비동기 해법의 경우는 TOL을 5로 선택한 경우 복구에 7.8초가 걸렸고 이는 전체 연산 시간의 4% 정도에 해당된다.

그림 5는 n_d=50번째 반복에서 장애가 발생할 경우, BkSL이 복구하여 연산을 재시작할 경우의 전체적인 수렴 속도를 나타내며, 두 번째 그림은 일부분을 확대한 것이다. 점선은 §3의 동기적 알고리즘이며, 1은 장애가 발생하지 않을 경우의 무정지형 비동기 알고리즘을 수행할 경우의 수렴을 나타낸다.



〈그림 5〉 n_d=50의 경우, 다양한 TOL 값에 따른 비동기적 알고리즘의 수렴

그림에서 그래프의 기울기가 가파를수록 수렴 속도가 빠름을 나타내는데, 동기 알고리즘의 결과에 비해 비동기 알고리즘의 수렴 속도가 약간 빠른 것을 볼 수 있으며, 장애가 발생한 후 복구된 이후는 장애가 발생하지 않은 경우보다 근소한 차이로 수렴속도가 개선되었음을 볼 수 있다.

n_d가 50으로 주어질 경우, 프로그램이 시작한 지 약 43초만에 장애를 일으키는 것으로 파악되었는데, 두 번째 그림으로 보아 장애가 발생한 이후에도 거의 50초까지 대략 8초 동안은 나머지 SL들의 연산만으로도 수렴이 진행되었고, 그 이후에는 복구될 때까지 수렴이 거의 진행되지 않았음을 볼 수 있다. 이는 비동기 반복법의 한 특징으로서, 만일 동기적 알고리즘을 사용하였다면 장애 발생 즉시 다른 슬레이브들은 연산을 계속할 수 없어 수렴이 중단되었을 것이다.

한편 MA에게 장애가 발생한 경우의 성능을 보기 위해, Algorithm 7에서 k_{MA}를 10의 값을 주고 실험한 결과는 표 1과 거의 동일한 결과를 얻었다. 그 이유는, BkMA는 항상 MA와 같은 데이터를 유지하도록 하였기 때문에 장애가 발생한 MA의 역할을 대신하는 데는 아주 짧은 시간(즉 MA가 SL들의 요구에 반응하던 시간의 k_{MA}배)으로 충분하기 때문으로 사료된다.

6. 결론 및 향후 연구 방향

본 논문에서는 무정지형 MPI 라이브러리를 사용하지 않고 MPI 표준함수와 비동기 연산을 사용하여 응용 프로그램 수준에서 무정지형 선형 시스템의 해법을 구현하였다. 특히 비동기 연산의 도입은 체크포인팅을 위한 별도의 데이터 전송의 필요성을 없애고 롤백의 필요성을 제거해 장애 복구 메커니즘을 단순화하였다.

성능 실험 결과, 제안한 알고리즘은 어느 노드에 장애가 발생되어 복구된 이후는 장애가 없이 계산이 진행되는 것에 비해 약간 수렴속도가 개선되었으며, 장애가 발생하지 않더라도, 동기 알고리즘에 비해 약간 수렴이 빠름이 판명되었다. 또한 장애가 발생하여 다른 프로세서가 이를 대신하여 복구를 하기 전이라도 어느 정도까지는 나머지 프로세서의 연산만으로도 계속 수렴이 진행되는 것이 관측되었다.

한편 본 성능 실험에서는 하나의 슬레이브 프로세서를 인위적으로 정지시켰는데, LAM MPI를 사용하여 실험한 경우 본 알고리즘은 이상없이 작동하였다. 그러나 운영체제 및 MPI의 종류에 따라, 예기치 않게 프로세스가 죽는 경우에도 제대로 작동할 지는 알 수 없고, 다른 MPI 환경에서의 실험 및 보완이 필요할 것으로 사료

된다. 또한 Algorithm 3에서 슬레이브는 매스터와 백업 매스터에게 각기 데이터를 보내도록 하였으나, 각 슬레이브를 매스터, 백업 매스터와 함께 각기 다른 커뮤니케이터를 형성하면 두 번의 데이터 전송을 하나의 브로드캐스터 메시지로 처리할 수 있을 것이다.

참 고 문 헌

[1] MPI Forum. 1995. MPI: A Message-Passing Interface standard.

[2] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovsky, & J. J. Dongarra, "Fault tolerant communication library and applications for high performance computing", Proceedings of the Los Alamos Computer Science Institute Symposium 2003, Santa Fe, NM.,

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

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 박필성, 신순철, "비동기 알고리즘을 이용한 분산 메모리 시스템에서의 초대형 선형 시스템 해법의 성능 향상", 한국정보처리학회 논문지 8-A(4):439-446, 2000.

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

◐ 저 자 소 개 ◑

박 필 성 (Pil Seong Park)

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