碩士學位論文

전기단층촬영을 위한 클러스터 기반 자코비안 성능향상 기법

濟州大學校 大學院

電算統計學科

孫 秀 昉

2002 年 12 月

전기단층촬영을 위한 클러스터 기반 자코비안 성능향상 기법

指導敎授 李 政 勳

孫 秀 昉

이 論文을 理學 碩士學位 論文으로 提出함

2002 年 12 月

孫秀昉의 理學 碩士學位 論文을 認准함

審査委員長 _____ 印

委 員 _____ 印

委 員 _____ 印

濟州大學校 大學院

2002 年 12 月

A performance enhancement scheme for Jacobian matrix via cluster computing on ET image reconstruction procedure

Su-Bang Son

(Supervised by Professor Jung-Hoon Lee)

A thesis submitted in partial fulfillment of the requirement for the degree of Master of Science

2002. 12.

Department of Computer Science And Statistics

GRADUATE SCHOOL

CHEJU NATIONAL UNIVERSITY

목 차

표 목 차

그 림 목 차

Abstract

This thesis proposes and measures the distributed Jacobian computation scheme based on the LINUX-based cluster technology, aiming at improving the image reconstruction speed of ET(Electronic Tomography) system which requires very intensive computation time, in spite of relatively low cost compared with other technologies. As an instance of the tomography technology, ET partitions the cross-section of the target object into the tiny elements and then computes their resistivity values, considering signal values measures at the boundary electrodes surrounding the surface of the object after injecting the predetermined current pattern into the object. In this procedure, Jacobian matrix calculation, which is analogous to the solution of partial differential equation systems, overwhelms the image reconstruction steps. Hence, a distributed algorithm is proposed after the careful analysis of the matrix representation policy of the numerical library as well as the data dependency of the procedure. It appears an instance of master-slave paradigm, where all member nodes compute in parallel the job given from the master. It is natural that each node loads static data structures before it can receive its mission.

The Jacobian process for ET, originally implemented and published by the researchers in this area, consists of a number of loops, and the respective loop calculates the Jacobian column, where the number of columns is identical to that of partitioned elements. The code analysis shows that each loop has no data dependency and that nothing but a small amount of data should be transmitted for distributed cooperative computing. To this end, we build a cluster composed with three PC's interconnected via hub-based local network. Each node runs or is stuffed with LINUX operating system, MPI(Massage Passing Interface), and numerical libraries. MPI. the standard message passing scheme on cluster enables each node to exchange messages carrying the parameter matrix, ρ, which is the intermediate vector of elements, as well as J, which is the partial Jacobian result produced at each node. The numerical library,

which may be Netlib, LAPACK, Matlab, and so on, provides an efficient matrix computing speed to calculates the partial Jacobian. We have implemented all codes with C programming language, also calling MPI and numerical library form the code. It is important to correctly specify the data block where the real data elements are located so that the distributed implementation can send a matrix via MPI library, as the numerical library represents a matrix of their own way. This environment supports expandability as long as the attached node has the same component as the others. In addition, the implementation can fulfil the Jacobian matrix operation regardless of the number of node in the cluster. Though a load balance scheme is not considered up to now, it can be exploited into our cluster.

The performance of the distributed Jacobian implementation according to the distributed scheme has been measured with 2 node cluster, since one node has very poor computation speed due to old-style CPU as well as small memory size. The experiments includes the computation time of matrix multiplication, that of Jacobian computation according to the variable allocation of columns. Matrix multiplication shows worse computation time than single node execution as it requires a great amount of data exchange between the node and the network speed of our cluster is limited to 10 Mbps. The distributed Jacobian shows better performance by 18 %, when the load is partition appropriately, that is, when the high performance master node computes more columns.

Consequently, the proposed distributed Jacobian can improve the image reconstruction time of ET system, exploiting the cluster framework built on our research. In addition, the computation speed can be further enhanced via the expansion of node, network performance upgrade, and adoption of a load balancing algorithm.

I. 서 론

ET(Electric Tomography), 즉 전기적 단층촬영 기법은 대상이 되는 물체에 인위적인 전기신호를 주입한 후 그 물체에서 나오는 신호를 분석하여 물체의 내부를 영상화한다(Cheney, 1999). 이 기법은 CT(Computerized Tomography)나 NMR(Nuclear Magnetic Resonance, MRI-CT) 등에 비하여 가격이 저렴하고, 또한 데이터의 취득속도가 빠르기 때문에 최근 새로운 단층촬영 기법으로 주목을 받고 있다. 그러나 영상 복원 과정에 있어서 계산 량이 막대하여 특별한 하드웨어의 도움 없이 일반적인 소프트웨어 산술계산(numerical computing) 도구를 이용하여 복원 알고리즘을 수행하는 경우 상당한 시간이 소요되므로 병렬 혹은 분산 처리에 의한 계산 속도의 개선을 필요로 한다(Wadleigh, 2000). 이러한 병렬 계산의 필요성을 충족시키기 위해 MPI(Message Passing Interface)나 PVM(Parallel Virtual Machine)과 같은 미들웨어들이 개발되어 있을 뿐 아니라 네트워크 기술의 발달에 따른 데이터 전송속도의 향상은 클러스터 컴퓨팅과 같은 병렬 분산 계산을 가능하게 한다(Buyya, 1999).

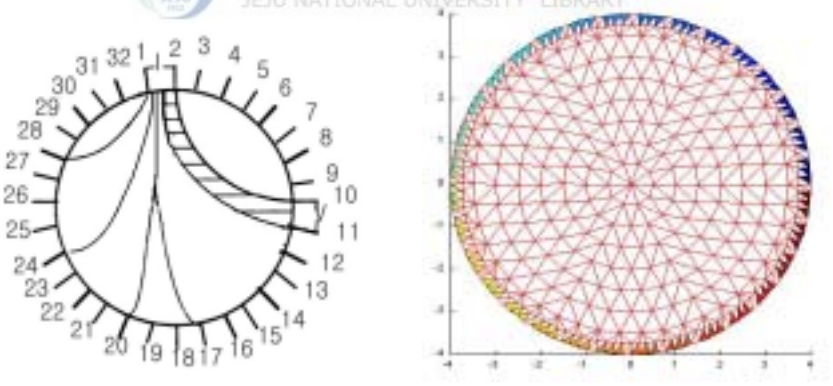
본 논문은 ET 영상 복원의 속도를 향상시키기 위하여 네트워크로 연결된 PC들이 병렬 계산을 수행하는 계산 환경을 구축하고 이를 기반으로 영상 복원 과정 중 가장 많은 CPU 시간을 필요로 하는 자코비안 계산에 대해 병렬 계산 응용을 작성하고 이의 성능을 측정한다. 먼저 계산 환경 구축을 위해 리눅스 운영체제를 탑재한 PC들을 하나의 네트워크로 연결시키고 각 PC들에 효율적인 산술계산 라이브러리와 MPI를 설치한다. 자코비안 계산은 요소의 수만큼의 반복 루프로 구성되고 각 루프는 상당한 양의 행렬 계산을 수행하지만 각 루프의 데이터 의존성이 없으므로 효율적으로 여러 노드에 분할되어 수행될 수 있다. 이 과정에서 MPI 미들웨어는 협력계산에 필요한 자료구조들이 각 노드 간에 효율적으로 교환될 수 있는 환경을 제공한다. 각 노드에 작업이 분할되면 각 노드는 할당된 계산을 수행하여야 하는데 산술계산 라이브러리는 전치(transporse),곱과 같이 필요한 행렬 계산을 효율적으로 수행한다(이장우, 2000).

본 논문의 구성은 다음과 같다. 1 장에서 문제를 제기한 후, 2 장에서는 ET 문제의 특성과 연구 배경에 대해 소개한다. 3 장에서는 클러스터 및 소프트웨어의 구축 내역과 프로그램 환경을 설명한다. 4 장에서는 자코비안 계산 과정을 제시한 후 클러스터 환경에서의 구현 내역을 자세히 설명한다. 5 장 에서는 수행시간을 측정한 결과를 보이고 마지막으로 6 장에서는 논문을 정리하고 결론을 도출한다.

II. 관련 연구

1. ET의 문제와 특성

ET는 <Figure 1(a)>에서 보는 바와 같이 물체 외부에 배치된 전극을 통해 전류를 순차적으로 주입하고, 각 주입된 전류에 대해 다른 전극에서 전압을 측정한다. 이 과정에서 <Figure 1(b)>에서처럼 물체 내부를 가상적인 요소(element)들로 분할한 후 측정된 전압을 기준으로 각 요소들의 저항 값을 계산하여 물체 내부의 특성을 파악한다. 각 요소들이 미지의 변수가 되며 전극에서 측정된 데이터들은 변수의 값을 계산하는데 필요한 방정식을 제공한다. 전극의 수가 많아지면 방정식이 많아지고 이에 의해 변수 개수를 늘일 수 있다. 결국 요소의 개수는 계산 시간에 큰 영향을 주며 외부 전극의 개수와 해상도, 속도 요구사항에 의해 결정된다.



(a) 외부 전극 (b) 메쉬 요소

Figure 1. The principle of Electronic Tomography

Netown-Raphson, Kalman Filter와 같은 영상 복원 알고리즘은 반복적인 루프로 구성되며 각 루프에서는 하나의 전극에서 주입된 신호에 의해 다른 전극들에서 측정된 신호, 즉 한 프레임의 데이터에 각 요소의 값을 계산한다(Bozic,

2000). 이 과정은 <Figure 2>에서 보는 바와 같이 FEM (Finite Element Method), 자코비안, 차이보정 등의 과정을 포함하는 복원 루프의 반복으로 구성된다. FEM은 현재 불완정하게 추정된 내부 요소들의 저항 값들에 의해 사전에 정의된 전류의 패턴이 주입되었을 때 예상되는 각 전극에서의 측정치를 계산하는 과정이며 실제 측정치와 예상치의 차이가 다음 복원 루프에 반영이 된다. 자코비안은 각 요소의 저항율을 순서대로 변화시켜 가면서 이에 대응하는 경계 전압의 변화를 계산하여 경계 전압의 변화와 저항율의 변화의 비를 계산함으로써 구할 수 있다. 이때 각각의 전극에 대해 모든 전류 주입 패턴, 즉 모든 FEM 요소를 대상으로 수행된다.

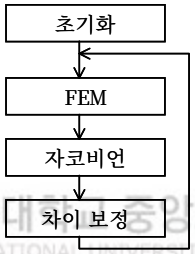


Figure 2. Overview of image reconstruction algorithm

한 반복 루프에서 처리되는 데이터들은 요소의 수에 의해 결정되는 다양한 행렬에 대한 연산으로 구성되어 있다. Pentium III PC, 128M 메모리, 윈도우용 MATLAB 5.3 환경에서 작성된 프로그램의 수행시간을 분석하면 한 루프가 평균 50초 정도 소요되고 이중 자코비안 계산이 33초로서 전체 반복루프의 가장 큰 부분을 차지한다. 따라서 자코비안 계산의 속도를 개선하는 것이 가장 전체적인 복원 속도 향상에 필수적이다.

2. 클러스터 컴퓨팅

1) 개요

수치계산 성능을 향상시키기 위한 다양한 방법이 존재한다. 그중 고성능 단일 컴퓨터를 이용한 계산은 이미 그 한계가 있음이 증명된 상태이며 이것에 대한 대안으로 다수의 프로세서(CPU)가 하나의 문제를 협동적으로 계산하는 병렬컴퓨팅이 등장하게 되었다. <Figure 3, 4, 5>에서 보이는 바와 같이, 다수의 프로세서가 하나의 메모리를 공유하는 SMP (Symmetric Multiprocessing) 플랫폼, 다수의 프로세서가 각각 독립된 메모리를 가지고 있는 MPP (Massively Parallel Processing) 플랫폼, 다수의 프로세서가 지역 메모리를 계층적으로 공유하는 NUMA (Non-uniform Memory Access) 등이 여기에 속한다. 그러나 기존 몇몇 업체에 의해 제공되던 병렬컴퓨터 혹은 슈퍼컴퓨터들은 매우 고가이기 때문에 쉽게 접하기가 어렵다(Almasi, 1988).

한편 최근 마이크로프로세서들은 뛰어난 성능을 보여주고 있으며 고속 네트워크 또한 일반화되고 있다. 이로 인해 단일 컴퓨터들을 네트워크로 연결함으로써 새로운 개념의 병렬컴퓨터를 만드는 것이 가능하게 되었다. 또한 리눅스 운영체제는 강력한 네트워크 성능을 제공하며 소스공개로 인한 자유로운 튜닝이 가능하기 때문에 이들을 위한 운영체제로 널리 사용되고 있다.

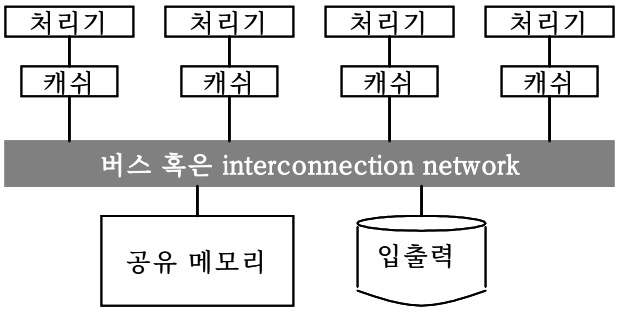


Figure 3. SMP(Symmetric Multiprocessor)

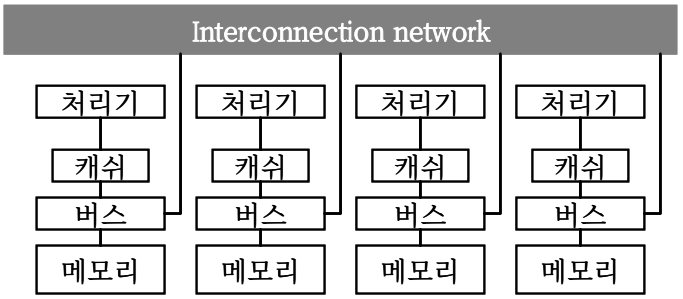


Figure 4. MPP(Massively Parallel Processor)

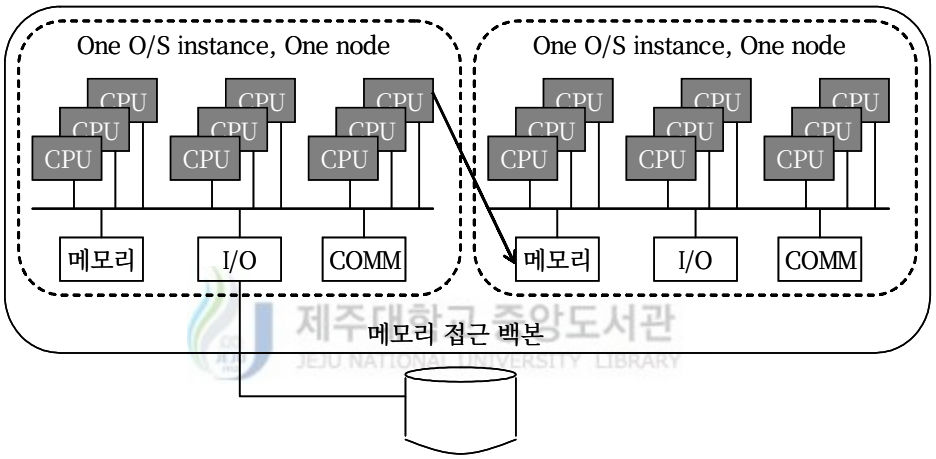


Figure 5. NUMA(non-uniform memory access)

이러한 개념의 병렬컴퓨터를 통칭하여 '클러스터'라고 부른다. 클러스터로 구성된 컴퓨터가 수퍼컴퓨터와 대등한 성능을 발휘하기 위해서 여러 대의 컴퓨터가 단일 컴퓨터로 동작하도록 관리해야 하며, 각 노드간에 데이터 교환이 가능하도록 설정해야 한다(Jordan, 2001).

2) 클러스터의 필요성 및 장점

많은 수치모델들은 좀 더 고품질의 결과를 얻기 위해서, 또는 빠른 결과를 얻기

위해서 대규모 계산을 수행해야 하거나 대규모 데이터를 처리해야 한다. 이를 위해 주로 워크스테이션이 활용되고 있으나 단일 워크스테이션으로는 충분한 성능을 제공받지 못하며 잠재적으로 성능 부족을 느끼게 된다(Baker, 2000). 병렬 컴퓨터는 이의 해결방안으로 좋은 대안이 될 수 있으며 특히 클러스터는 다음과 같은 주목할 만한 장점을 갖는다. 첫째, 워크스테이션에 비해 월등한 컴퓨팅 성능을 제공할 수 있으며 대규모 데이터 처리가 가능하다. 둘째, 범용하드웨어를 사용함으로 인해 상용 병렬컴퓨터에 비해 가격대 성능비가 매우 우수하다. 셋째, 노드의 증설에 따라 성능향상이 자유로우며 비교적 성능이 떨어지는 컴퓨터들을 이용하여 고성능 병렬 컴퓨터를 제작할 수 있다. 넷째, 자체 제작이 가능기 때문에 문제 발생시 자체 해결이 용이하다.

3) 클러스터의 종류

클러스터의 분류를 정의하는 방법에는 매우 다양하다. 일반적으로 많이 사용되는 분류법은 클러스터의 그 활용 목적에 따라 분류하는 것이며 이에 따르면 아래의 3가지로 나눌 수 있다. 고성능 클러스터 (HPC ; High Performance Cluster Computing), 부하분산 클러스터 (LVS ; Linux Virtual Server), 그리고 고가용성 클러스터 (HA ; High Availability)이다.

(1) 고성능 클러스터

HPC 클러스터는 고성능의 계산능력을 제공하기 위한 목적으로 제작되는데 주로 과학계산용으로 활용가치가 높다. HPC 클러스터를 구성하는 모든 컴퓨터들은 네트워크에 연결되어 있어서 상호간에 통신이 가능하므로 다수의 프로세서가 협동적으로 문제를 풀 수 있는 환경을 제공하게 된다. 그러나 이 기반을 이용하여 문제를 병렬로 푸는 것은 전적으로 프로그램 수준에서 이루어진다. 따라서 클러스터 구축뿐만 아니라 프로그램 병렬화가 같이 고려되어야 한다.

(2) 부하분산 클러스터

대규모의 서비스를 제공하기 위한 목적으로 제작되며 주로 웹서비스 등에 활용가치가 높다. <Figure 6> 은 부하분산용 클러스터의 개념도이다. 이러한 방식은 인터넷 사용자로 하여금 하나의 컴퓨터 즉 로드밸런스로부터 서비스를 제공받는 것으로 느끼게 한다. 그러나 실제 서비스 제공은 로드밸런서에 연결되어 있는 서비스 노드들에 의해서 이루어진다. LVS 클러스터 역시 일반적으로 모든 컴퓨터간에 연결되어져서 구성된다. 그러나 이것은 필수적인 조건은 아니다. 만약 정적인 데이터만을 서비스하는 웹 클러스터라면 각 서버들은 Load Balancer와의 통신만 가능해도 된다. 또는 그 중간적인 형태 역시 가능하다. 즉 클러스터의 위상(topology)에 있어서는 다양한 형태로의 변형도 가능하다. LVS 클러스터의 웹 요청 처리과정을 살펴보면 다음과 같다.

- 로드밸런서에게 인터넷으로부터 웹 요청이 들어온다.
- 로드밸런서는 정해진 알고리즘에 따라 서비스를 수행할 노드를 선택하고 웹 요청을 포워딩 한다. 포워딩된 웹 요청은 선택된 서버로 이송된다.
- 웹 요청을 받은 서버는 이에 대한 응답을 로드밸런서에게 제공한다.
- 로드밸런서는 제공받은 데이터를 웹 요청을 한 컴퓨터로 전송해준다.

부하분산용 클러스터 중에서도 이러한 방식을 NAT(Network Address Transaction)라고 한다. 이 방식의 장점은 서버가 어떤 운영체제이든지 상관이 없다는 것이다. 그리고 IP Tunneling 방식은 웹 요청이 들어오면 요청패킷을 캡슐화해서 클러스터내의 노드들에게 전송해준다. 이것은 응답이 로드밸런서를 거쳐서 나가지 않으므로 하나의 로드밸런서가 매우 많은 서버들을 거느릴 수가 있다. 그러나 클러스터내의 서버들이 캡슐화된 패킷을 해석할 수 있어야 하므로 현재는 리눅스에서만 가능하다. 마지막으로 DR (Direct Routing)방식은 클러스터의 확장성을 높이기 위해 IP-NAT 방식과 IP-Tunneling방식의 장점만을 가져온 방식이다. 즉 서비스요청이 들어오면 패킷을 최소한으로 빨리 가공하여 실제 서버에게 보내면 실제 서버는 다른 경로를 통하여 응답을 보내는 것이다. 라우터가 요청패킷에 MAC 어드레스를 추가하여 유일한 실제 서버를 결정할 수 있게 해

준다. 이것을 위해서는 모든 노드들이 단일 세그먼트에 존재해야한다. 이 시스템의 단점으로는 로드밸런스의 고장이나, 기타 여러 다른 문제가 발생하면 서비스가 원활하게 제공될 수 없다.

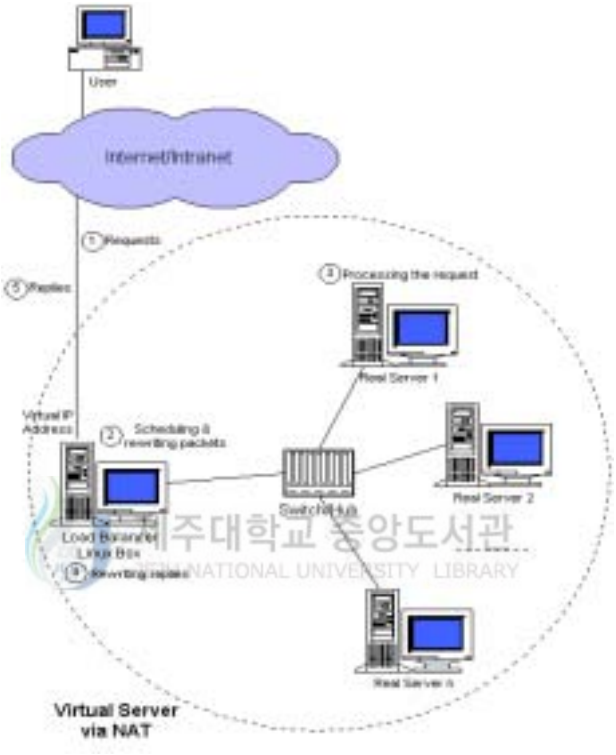


Figure 6. Load distribution cluster

(3) 고가용성 클러스터

지속적인 서비스 제공을 목적으로 제작되며 주로 중요한 작업 업무에 사용된다. 위에서 설명한 LVS 클러스터의 경우 만약 로드밸런서가 고장 났다면 클러스터 전체의 동작이 불가능하다. 또한 서버들 중 서버 2 에 고장이 발생하는 경우 일부 사용자는 원하는 데이터를 제공받지 못할 것이다. 이러한 문제를 극복하기 위한 목적으로 구성하는 것을 고가용성 클러스터라고 한다.

로드밸런서의 고장에 대처하기 위해 현재 사용되는 방법은 heart beat 와 fake를 이용하는 방법이다. <Figure 6>을 보면 heart beat는 로드밸런서와 하나의 백업서버간에 주기적으로 통신을 하며 이상 유무를 체크하게 된다. 로드밸런서의 고장이 인식되면 fake 라는 프로그램은 로드밸런서가 점유하고 있는 IP를 백업서버로 이주시켜서 계속 서비스를 수행시켜준다. 한편 서버 2 의 고장은 로드밸런서에 의해서 주기적으로 감시되고 고장 발생시 서비스 요청을 서버 2 로 포워딩 하지 않음으로써 서비스가 지속적으로 수행하게 된다.

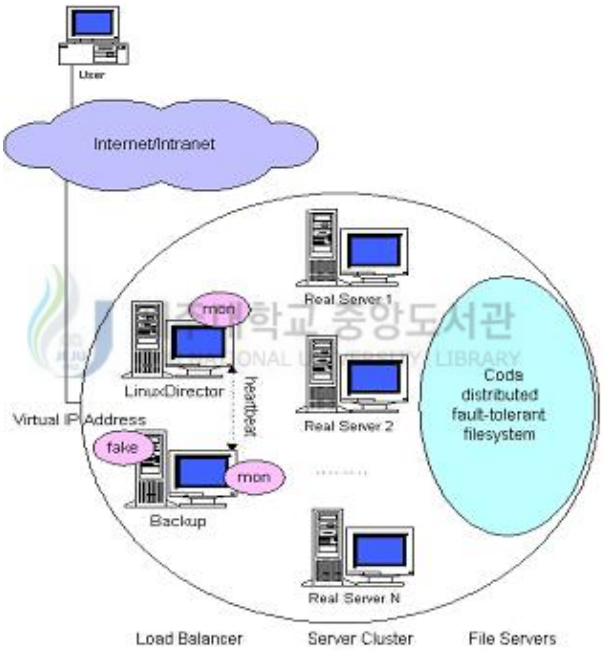


Figure 7. High availability of LINUX virtual server

(4) 기타 다른 클러스터 분류

기타 현재 여러 다양한 특성에 따라 다음과 같이 분류할 수 있다. 노드의 소유권에 따라서 (Node Ownership) 분류, 노드의 하드웨어에 따라서 node hardware 분류, 노드의 운영시스템에 따라서 node operating system 분류, 노드

의 구성에 따라서 node configuration 분류, 그리고 노드의 위치와 숫자에 따른 구분 level of clustering 등으로 할 수 있다.

4) 클러스터의 단점

클러스터의 문제점은 관리와 프로그램 작성의 어려움이다. 이러한 문제점은 다수의 컴퓨터로 구성되어 있다는 것에 기인한다. 즉 다수의 컴퓨터를 관리하고 일반적인 문제들을 해결하는데 상대적으로 많은 노력이 필요하다는 것이다. 다행히 관리의 어려움을 해결하기 위한 많은 도구들이 공개되어 있으며 이러한 도구들을 이용하여 상당부분 관리에 필요한 노력을 줄일 수 있다(Libertone, 2000). 한편 기존 워크스테이션에서 수행되는 프로그램들은 단일 컴퓨터에서 수행되도록 프로그램되어 있다. 이를 시리얼 프로그램이라 한다. 그러나 클러스터를 비롯한 병렬 컴퓨터에서는 병렬 프로그램을 사용하여야만 한다. 인터넷을 통하여 병렬화된 프로그램을 구할 수도 있지만 공개된 병렬프로그램이 없다면 직접 프로그램을 병렬화하여야 한다. 이것은 경우에 따라 매우 많은 노력을 필요로 한다.

3. 부하균배

1) 개념

협력 작업을 위해서는 주어진 작업을 분할하여 클러스터내의 각 노드에 할당하여야 하는데, 각각의 작업을 할당하는 방법은 크게 정적인 방법과 동적인 방법 2가지로 나뉜다. 정적인 방법은 사전에 정해진 순서에 따라 수행 하는 방법이며, 동적인 방법은 관리 서버가 각각의 클라이언트를 모니터링해서 현재의 부하를 파악하여 적절한 컴퓨터에 작업을 할당하는 방법이다. 로드밸런스는 클러스터로 연결된 노드에 적절하게 작업을 분배, 관리해야 한다. 여기에는 각 컴퓨터 상태 즉, 통

신 프로토콜, 메모리, 기타 여러 상태를 파악할 수 있어야 한다. 여러 대의 컴퓨터가 서로 연관해서 작업을 하기 때문에 많은 문제가 생길 수 있다(Bung, 1999).

2). 기본적인 밸런싱 방법들

가중치(Weighting) 방법은 단순히 우선순위를 정해서 가중치가 높은 것을 먼저 사용하게 하는 방법으로, 일단 사용되는 것은 우선순위를 낮게 하여 다른 것을 높임으로써 각 노드의 다음 상태를 결정 할 수 있으며 네트워크의 부하를 줄이는 것이다. 랜덤화(Randomization) 방식은 가상 랜덤 알고리즘을 이용하는 방법으로서 높거나 낮은 랜덤 값을 정해서 노드에 할당을 한다. 라운드로빈(Round-Robin) 방식은 Rotary, 또는 Cyclic 방법이라고 불리며 이것은 단순히 나열된 노드의 리스트 순서대로 순위를 정하는 것이다. 이 방식은 DNS 이름 찾기에서 사용되어 진다. 또한 이 방식은 클러스터가 모두 접근이 가능할 때 효과적이다. 해싱(Hashing) 기법은 같은 목적지 주소에서의 패킷이 같은 서버에 할당되는 단순 가중치 시스템과 유사하다. 즉 같은 소스 주소의 패킷은 같은 서버에 할당이 된다. 또한 보다 높은 세션에서 사용가능 하다. 최근 연결(Least Connections)은 현재의 모든 연결을 유지하고 다음 새로운 연결은 현재 최소의 연결은 갖는 노드를 요구한다. 이 방법의 문제점은 많은 리소스 자원을 소비하는데 있다. 최소실패(Minimum Misses)는 밸런싱 디바이스가 노드에 들어오는 요청할당을 장시간 유지하고, 다음 요구를 히스토리(history) 정보에서 가장 적게 요구된 노드에 할당한다. 빠른 응답(Fastest Response)은 노드와 그 자신 그리고 가장 빠른 응답을 가지는 노드에 대해 응답 시간을 지속한다. 이것은 노드들에 ICMP 패킷을 보내서 지속적으로 관찰을 한다.

3) 진보적인 밸런싱 방법들

이 방법은 단일적으로 이용되는 경우 보다는 서로 조합해서 이용을 한다. 단순 방법을 조합한 것으로 대표적인 방식은 다음과 같다.

■ 네트워크 트래픽에 기반을 둔 밸런싱(Network Traffic-based Balancing)

이 시스템은 들어오는 트래픽 활동을 모니터링 한다. 또한 이 기술은 클러스터에 독립적이며, 각각의 노드가 동일하거나 비슷할 때 좋은 성능을 낸다. 이 방법은 네크워크 트랙픽 최적화, 네트워크 라우터 최적화, 응답 대기 시간 최소화를 제공한다.

■ 노드 트래픽에 기반한 밸런싱(Node Traffic-based Balancing)

네트워크 트래픽 밸런싱 시스템 (Network traffic balancing system)과 반대 개념이다. 주요 방법은 최근 연결, 최소 실패, 빠른 응답을 이용한다. 랜카드의 영향을 많이 받는다.

■ 노드 로드에 기반한 밸런싱 (Node Load-based Balancing)

밸런싱 장비가 소프트웨어 에이전트를 관리한다. 밸런싱 장비는 어떠한 단순 밸런싱 방법도 이용 할 수 있다.

4) 일반적인 에러들(Common Errors).

일반적인 에러에는 오버플로우, 언더플로우, 라우팅 에러, 네트워크 에러가 있다 오버플로우는 노드가 처리할 수 있는 능력보다 더 많은 패킷을 받을 때 발생하는데 밸런싱 장치 또는 각 노드에서 발생할 수 있다. 언더플로우는 노드 자신의 문제로서, 클러스터로 연결된 다른 컴퓨터가 비교해서 충분한 데이터를 받지 못하는 경우이다. 라우팅 에러는 환경 설정 에러나 연결이 끊어질 경우 발생한다. 야기된 네트워크 에러는 이것은 확실한 에러가 아니라 네트워크 경로 상에서 패킷이 딜레이로 인한 결과이다.

5) 실제 구현(Practical Implementations)

Holon, Cisco, Resonate, Sun 등 많은 제작자들이 다른 각도에서 장비를 제작하지만 해결 방법은 비슷하다. 각각의 장비회사들이나 응용 제작회사 들은 기존의 여러 방법들을 응용 혼합해서 제품을 만든다.

4. MPI

1) 개념

한 개의 CPU가 가질 수 있는 계산능력에는 한계가 있으며 이 한계를 극복하기 위해서 여러 개의 CPU를 탑재할 수 있는데 이 방법은 많은 비용을 요구하며, 또한 개발에도 많은 어려움이 따른다. 이를 해결하기 위한 방법이 여러 대의 컴퓨터를 네트워크로 연결해서 협력 처리 하는 클러스터 컴퓨팅 기법이 나오게 되었다 (Lusk, 1997).

이 환경에서 병렬 응용을 작성하는데 필수적인 미들웨어로서 크게 MPI (Massage Passing Interface)와 PVM (Parallel Virtual Machine)이 출현하였으며 이 둘의 차이점은 다음과 같다. PVM의 경우 메시지를 전송하기 전에 pack(일종의 압축)함으로서 서로 다른 이종간에 전송이 자유롭게 이루어진다. 이에 반해 MPI의 경우는 이러한 메시지 전송이 자유롭지 않지만 MPI_xx라는 동일한 형을 가짐으로 이종간에 전송이 가능하다. pack 하지 않으므로 얻는 장점은 전송 하는데 드는 시간을 절약할 수 있다.

클러스터에서 병렬로 문제를 풀기 위해서는 독립적인 컴퓨터간의 통신이 필요하게 된다. 이는 주로 네트워크를 통하여 이루어지게 되는데 방정식(equation) 해 찾기와 같이 데이터 혹은 프로그램 제어구조의 독립구조의 독립성이 강한 응용에서 네트워크 프로그램을 다루는 것은 매우 비효율적이다. 그래서 컴퓨터들 간의 메시지를 편리하게 주고받기 위해 많은 병렬 라이브러리들이 등장하게 되었고, 이중 MPI가 국제표준으로 선정되었다.

MPI 자체는 병렬 라이브러리들에 대한 표준규약이다. 따라서 MPI를 따르는 병렬 라이브러리를 사용한다면 소스 레벨의 호환성을 보장받는 장점이 있다. 또한 MPI는 약 40개 기관이 참여하는 MPI 포럼에서 관리되고 있으며, 1992년 MPI 1.0을 시작으로 현재 MPI 2.0까지 버전업된 상태이며 이들 MPI를 따르는 병렬 라이브러리로는 오하이오 슈퍼컴퓨터 센터에서 개발한 LamMPI 와 Argonne National

Laboratory에서 개발한 MPICH가 널리 사용되고 있다.

2) MPI 데이터타입과 MPI 연산타입

MPI 프로그램에서는 데이터를 전송 또는 수신할 때 반드시 데이터 타입과 함께 전송해 주어야 한다. 모든 데이터는 바이트 형태로 전송되므로 이를 원래의 데이터로 해석하기 위해서는 데이터 타입을 반드시 알아야 한다. 따라서 거의 모든 데이터 타입에 대해 <Table 1>과 같이 정의가 되어 있다.

Table 1. MPI Datatype

MPI C 데이터 타입	
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI_Reduce() 함수와 같은 몇몇 함수들은 각 프로세스로부터 데이터를 모아서 일정한 연산을 수행한 후 결과를 돌려주게 된다. 이를 위해서 연산타입이 아래와 같이 정의 되어 있다.

Table 2. MPI Operation Type

MPI Reduction Operation		C data types
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double, long
MPI_MINLOC	min value and location	float, double, long

3) MPI의 함수

(1) 기본 함수

MPI에서는 120개가 훨씬 넘는 많은 함수들을 정의해 놓고 있다. 이들 대부분은 6개의 기본적인 함수들의 조합으로 구현될 수 있으며 이들 함수는 아래와 같다.

■ MPI_Init

프로그램 실행 인수들과 함께 MPI함수들을 초기화 해준다.

■ MPI_Finalize

MPI함수들을 종료한다. 모든 MPI함수들은 MPI_Init() 과 MPI_Finalize() 사이에서 호출 되어 진다.

■ MPI_Comm_rank

comm 커뮤니케이터에서 자신의 프로세스 아이디(id)를 얻는다.

3개의 기본 MPI_Comm 함수

- MPI_COMM_WORLD : 기본 그룹
- MPI_COMM_SELF : 하나의 멤버 또는 자기 자신에 대한 그룹.

- MPI_COMM_PARENT: 자신의 기본 그룹과 자신의 부모 그룹 사이에서 통신자.

■ MPI_Comm_size

comm 커뮤니케이터에서 실행되는 프로세스의 개수를 얻는다.

■ MPI_Send

dest로 메시지를 보낸다. message는 보내고자 하는 메시지를 저장하고 있는 버퍼이다. count 는 보낼 메시지 개수, datatype는 보낼 메시지 타입이다. dest는 보내고자 하는 프로세스 id이다. tag는 보내는 메시지에 대한 꼬리표이다. comm은 dest와 자신의 프로세스가 속해있는 커뮤니케이터이다.

■ MPI_Recv

source로부터 메시지를 받는다. message는 받은 메시지를 저장할 버퍼이다. count는 받을 메시지 개수(받는 메시지 개수보다 작으면 에러발생)이다. datatype은 받을 메시지 타입이다. source는 메시지를 보내주는 프로세스 id이다. tag는 받은 메시지를 확인하기 위한 꼬리표(MPI_Recv에서의 tag와 MPI_Send에서의 tag가 같아야한다)이다. comm은 source와 자신의 프로세스가 속해있는 커뮤니케이터이다. status는 실제 받은 데이터에 대한 정보(source와 tag)이다.

(2) 함수 확장

공통 통신 함수는 어떤 커뮤니케이터에 있는 모든 프로세스가 같이 모두 호출되어야 하는 함수이다. 커뮤니케이터란 프로세스 그룹을 말한다. 기본적으로 MPI프로그램에서는 MPI_COMM_WORLD라는 기본 커뮤니케이터가 생성된다. 이는 동시에 수행되는 모든 프로세스를 포함한다. 그러나 사용자는 임의의 프로세스로 구성된 새로운 커뮤니케이터를 생성할 수 있다. 이에 대한 것으로는 *MPI_Bcast, MPI_Reduce*, *MPI_Barrier* 등이 있다.

통신에 대한 그룹화 데이터 함수는 주로 여러 개의 데이터들을 모아서 한번에 전송하도록 해주는 함수들로 구성되어 있다. 이러한 함수들이 필요한 이유는 일반

적으로 빈번한 메시지 패싱 보다는 이들을 묶어서 한번에 보내주는 것이 효율 면에서 좋기 때문이다. 이에 대한 함수로는 *MPI_Type_Struct*, MPI_Type_commit. *MPI_Type_ contiguous* 등이 있다.

Communicators 와 Topologies 함수는 다양한 커뮤니케이터를 만들고 프로세서들의 위상을 선언할 수 있는 함수들로 구성된다. 이에 대한 함수로는 MPI_Comm_group, MPI_Comm_creat 등이 있다.

III. 클러스터 구축

1. 설정 내용

자코비안의 병렬 계산을 위한 클러스터는 3 대의 PC와 1 대의 스위칭 허브로 구성된다. 각 노드의 사양은 <Table 3>에서 보이는 바와 같으며 각 노드는 10 Mbps 이더네트 인터페이스를 통해 DAVID SYSTEM의 DSI 허브와 연결되어 있다. 지역 클러스터 구축을 위해 노드 1은 두 개의 네트워크 인터페이스를 갖고 있는데 하나는 외부로 접근할 수 있는 인터페이스인 반면 다른 하나는 클러스터 내의 노드들을 연결하고 있어서 클러스터를 외부의 트래픽과 차단시킨다. 결국 클러스터는 외부에서 하나의 노드로 인식된다. 또한 각 노드는 Redhat 리눅스 7.3 운영체제 상에 MPI lam-6.5.6와 산술계산 라이브러리를 탑재하고 있다. 이를 기반으로 C 언어로 자코비안 응용을 작성하여 수행시킬 수 있다. 물론 노드를 더 추가할 수 있으며 보다 효율적인 산술계산 라이브러리로 교환할 수 있다(이정훈, 손수방, 2002).

구성된 모습은 <Figure 8> 과 같다. Node 1은 마스터 노드이며 랜 카드는 2개로 구성되어 있다. 나머지 2개는 슬래이브 노드이다. 클러스터는 <Figure 9>와 같이 노드의 수를 확장할 수 있으며 관리 노드를 두는 방안이 바람직하다.

Table 3. Node Description

HOST	CPU(Intel)	대역폭(Mbps)	메모리
노드1	Pentium-III 933	100(2)	256 M
노드2	Pentium-II MMX 333	100	256 M
노드3	Pentium I 200	10	96 M

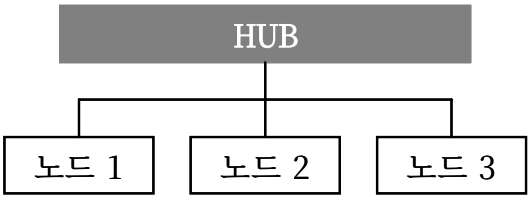


Figure 8. Implemented architecture

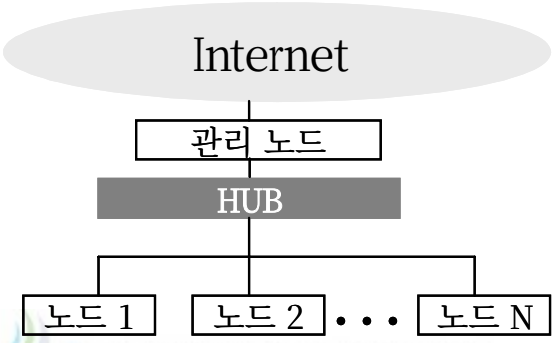


Figure 9. Target structure

1) 사설 네트워크 구축

구축된 클러스터는 3대의 컴퓨터가 하나의 독립된 네트워크를 형성하게 된다. 사설 네트워크를 구축하는 이유는 아래의 몇 가지를 들 수 있다. 첫째, 모든 사용자가 관리노드로 접속하여 작업을 하도록 유도함으로써 관리가 편리하다. 둘째, 모든 노드에 대한 보안설정이 필요 없고, 관리노드에 대한 보안설정만 하면 된다. 셋째, 컴퓨팅 노드들은 외부 패킷의 영향을 받지 않을 수 있다. 넷째, 불필요한 공인 IP의 낭비를 막을 수 있다.

2) 관리노드 세팅

Step 1 : 네트워크 설정하기

두 개의 랜카드에 다음과 같은 주소를 부여한다. eth0만 외부에서 인식되며, eth1 은 내부 네트워크에서 사용될 주소이므로 어떠한 값이 사용되어도 상관없다.

eth0 : IP(210.93.74.164) hostname : master.cheju.ac.kr

eht1 : IP(192.168.1.1) hostname : node1.cheju.ac.kr

Step 2 : hosts 파일 편집한다.

클러스터를 이루고 있는 각 노드의 주소와 이름을 정의해 준다. 이것은 관리노드와 모든 컴퓨팅노드에서 동일하게 설정해 준다.

210.93.74.164	master.cheju.ac.kr	master login
192.168.1.1	node1.cheju.ac.kr	node1
192.168.1.2	node2.cheju.ac.kr	node2
192.158.1.3	node3.cheju.ac.kr	node3

Step 3 : nfs 데몬을 활성화한다.

nfs는 RPC(Remote Procedure Call)를 사용하기 때문에 포트매퍼(Port mapper)라는 데몬이 먼저 떠 있어야 한다. 이 데몬이 수행되고 있지 않다면 쉘 상에서 setup 명령을 사용하여 nfs 데몬을 활성화 한 후 재시작 한다.

Step 4 : rsh을 허가한다.

클러스터내의 모든 노드들간에 rsh 이 가능하도록 세팅해야 한다. 이것은 MPI가 정상작동하기 위해 필요하다. 이 데몬이 활성화되어 있지 않다면 쉘 상에서 setup 명령을 사용하여 rsh를 활성화 한 후 재시작 한다. 그리고 어떤 컴퓨터들이 자신에게 rsh이 가능하도록 허가할지를 hosts.equiv에 정의하여야 한다. hosts.equiv 파일이 없으면 이를 만들고 호스트 이름을 적는다. 이제 이 두 개의 호스트 간에는 일반 사용자에 한해서 rsh 명령이 가능해 진다.

(5) 병렬라이브러리를 설치한다.

MPI 라이브러리에는 몇 가지 종류가 있다. 그중 유명한 것이 LamMPI 와 MPICH이다. 이 논문에 이용한 라이브러리는 LamMPI 이다. 설치를 끝낸 후 사용자가 LamMPI를 사용하기 위해서 LAMHOME 이라는 환경변수를 선언해야 한다. 이렇게 함으로써 모든 사용자가 로그인할 때 LAMHOME 과 PATH가 설정된다.

3) 컴퓨팅노드 세팅

node2와 node3는 단지 IP 및 호스트 주소만 다르고 나머지는 같은 방법으로 설정이 되었다. 관리노드와 마찬가지로 리눅스를 설치했다.

Step 1 : 네트워크 설정하기

하나의 랜 카드에 다음과 같은 주소를 부여했다. eth1에 부여되는 주소에 대해서는 내부네트워크에서 사용될 주소이므로 어떠한 값이 사용되어도 상관이 없지만, 여기서는 node2(192.168.1.2), node3(192.168.1.3)으로 부여했다.

Step 2 : hosts 파일 편집한다.

클러스터를 이루고 있는 각 노드의 주소와 이름을 “/etc/hosts” 파일에 관리 노드와 같이 정의 한다.

Step 3 : nfs 데몬을 활성화한다.

Step 4 : rsh을 허가한다.

Step 5 : 병렬라이브러리를 설치 및 테스트

라이브러리 설치는 마스터의 방법과 동일하다. 설치가 끝난후 정상적으로 동작을 하는지 테스트를 했다. 사용자로 로그인을 한 후 lamhost 파일를 만들었다. 여기에 MPI 라이브러리가 접근할 host 이름이 기록된다.

Step 6 : 호스트를 활성화한다.

환경 세팅이 정상적으로 된 후 명령어 상태에서 " lamboot -v lamhosts" 라는 명령으로 MPI 라이브러리를 활성화 했다. 그 후, 병렬 라이브러리를 포함하여 컴파일을 했다. gcc 대신 hcc를 사용해야 한다.

4) 클러스터의 확장

lamhost 파일에 나머지 호스트 주소를 넣어서 확장이 가능하다.

5) 설치 테스트 및 예제 프로그램 수행

MPI 프로그램에서 제공되는 아래의 소스를 MPItest.c로 저장한 후 아래와 같은 일련의 과정을 거쳐서 수행한 후 결과의 정확성을 확인하였다.

```
[iedo]$ lamboot -v lamhosts
LAM 6.3.2/MPI 2 C++ - University of Notre Dame
Executing hboot on n0 (node1)...
Executing hboot on n1 (node2)...
topology done
[iedo]$ hcc -o MPItest MPItest.c -lMPI
[iedo ]$ MPIrun -np 2 ./MPItest
message=process 0, source=0, tag=123
```

```
#include <stdio.h>
#include "MPI.h"
int main(int argc, char **argv)
{ int rank, size;
 char data[10];
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 if(rank == 0) {
    strcpy(data, "process 0");
  // MPI_CHAR 는 전송하는 데이터가 char 타입이라는 데이터 타입 선언이다.
  MPI_Send(data, 10, MPI_CHAR, 1, 123, MPI_COMM_WORLD);
 }
 else if(rank == 1)
 {
  MPI_Recv(data, 10, MPI_CHAR, 0, 123, MPI_COMM_WORLD, &status);
  printf("message=%s, source=%d, tag=
 }
 MPI_Finalize();
 return 0;
}
```

IV. 분산 자코비안 프로그램

1. 자코비와 가우스-사이델의 방법의 소개

일반적으로 가우스 소거법(또는 가우스-조르단의 소거법)은 n 변수를 갖는 n 개의 방적식으로 이루어진 연립 1차 방정식을 푸는 경우 선택하는 방법이나 어떤 경우에 있어서는 연립 1차 방정식을 푸는 데 더욱 좋은 반복법(intertive methods) 또는 간접법 (indirect methods)이라 하는 다른 방법이 있다(김규철). 이들 방법은 해의 초기 근사값에서 출발하여 정확한 해로 접근하는 점점 좋은 일련의 근사값을 생성한다. 이 방법들은 18세기 후반으로 거슬러 올라가는 고전적인 방법이나, 오늘날에도 행렬이 크고 많은 수의 원소가 예측할 수 있는 위치에 있는 행렬이 수반되는 문제에 적용된다. 예를 들면, 이런 형태의 문제들은 큰 집접회로의 연구나 경계값 문제나 편미분 방정식의 수치해석에 흔히 나타난다. ET 영상복원에서도 자코비안에 기반하여 내부영상을 계산한다.

자코비 반복법(Jacobi iteration) 또는 동시반복법(method of simultaneous displacement)이라 하는 가장 간단한 반복법에 대해서 우선 먼저 설명한다. 이 방법을 n 변수의 n 개 방정식을 갖는 연립 1 차방정식에 일반적으로 적용이 된다. 연립방정식

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \mathrm{L} \;\; + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \mathrm{L} \;\; + a_{2n}x_n &= b_2 \\ \quad \mathrm{M} \qquad \mathrm{M} \qquad \mathrm{M} & \\ a_{n1}x_1 + a_{n2}x_2 + \mathrm{L} \;\; + a_{nn}x_n &= b_n \end{aligned} \qquad \text{<Eq. 1>}$$

은 틀림없이 해를 가지며 또한 계수행렬의 대각성분 $a_{11}, a_{22}, \cdots, a_{nn}$ 은 어느 것이나 0 이 아니라 하자. 첫째로 <Eq. 1> 의 첫째 방정식에서 x_1을 나머지 변수로서 나타내고 다음에 둘째 방정식에서 x_2를 그 나머지 변수로 나타낸다. 다음 셋째 방

정식에서 x_3를 나머지 변수에 대해서 나타낸다. 이것을 반복하여

$$x_1 = \frac{1}{a_{11}}\left(b_1 - a_{12}x_2 - a_{13}x_3 - \mathrm{L}\ - a_{1n}x_n\right)$$
$$x_2 = \frac{1}{a_{22}}\left(b_2 - a_{21}x_1 - a_{23}x_3 - \mathrm{L}\ - a_{2n}x_n\right)$$
$$\mathrm{M} \qquad \mathrm{M} \qquad \mathrm{M}$$
$$x_n = \frac{1}{a_{nn}}\left(b_n - a_{n1}x_1 - a_{n2}x_2 - \mathrm{L}\ - a_{nn-1}x_{n-1}\right) \qquad \text{<Eq. 2>}$$

을 얻는다. 예컨대 주어진 연립방정식을

$$20x_1 + x_2 - x_3 = 17$$
$$x_1 - 10x_2 + x_3 = 13$$
$$-x_1 + x_2 + 10x_3 = 18 \qquad \text{<Eq. 3>}$$

이라 하면,

$$x_1 = 0.850 - 0.05x_2 + 0.05x_3$$
$$x_2 = -1.3 + 0.1x_1 + 0.1x_3$$
$$x_3 = 1.8 + 0.1x_1 + 0.1x_3 \qquad \text{<Eq. 4>}$$

로 된다. 이제 <Eq. 1>의 근사해가 하나 얻어졌다 하면 이들을 <Eq. 2>의 우변에 대입해서 새로운 $x_1, x_2, \cdots, x_n$ 을 만든다. 이 경우 처음 근사해 보다도 더욱 좋은 근사해가 되어 있음을 알 수 있다. 이 사실이 자코비 반복법에 대한 열쇠이다.

자코비의 반복법에 의하여 <Eq. 1>을 풀려면 하나의 근사해를 만들어 둘 필요가 있다. 좋은 초기값을 예측하기 어려운 경우에는 $x_1=0, x_2=0, x_3=0, \cdots$ 이라 보통 정해서 한다. 이를 <Eq. 2> 의 우변에 대입해서 새로운 $x_1=\frac{b_1}{a_{11}}, x_2=\frac{b_2}{a_{22}}, \cdots, x_n=\frac{b_n}{a_n}$ 이 얻어지는 데 이것을 제 1 근사해가 된다. 또한 이것을 <Eg 2>의 우변에 대입해서 이하 이것을 반복하면 된다. 예컨대 <Eq. 3>의 제 1 근사해를 얻기 위해서 $x_1=0, x_2=0, x_3=0$ 이라 놓고서 이것을 <Eq.

4>의 우변에 대입하여 새로운 근사해,

$$x_1=0.850, \qquad x_2=-1.3, \qquad x_3=1.8 \qquad \text{<Eq. 5>}$$

을 얻는다. 이 근사해를 개선하기 위하여 대입과정을 반복할 수 있다. 예컨대 <Eq. 3>을 푸는 경우 <Eq. 5>를 또한 <Eq. 4>의 우변에 대입해서 제 2 근사해를 구하면,

$$x_1 = 0.850 - 0.05(-1.3) + 0.05(1.8) = 1.005$$
$$x_2 = -1.3 - 0.1(0.850) + 0.1(1.8) = -1.035$$
$$x_3 = 1.8 - 0.1(0.850) - 0.1(-1.3) = 2.015$$

이다. 어떤 조건을 만족하는 경우에는 이와 같은 근사방법을 반복해서 얼마라도 이 연립방정식의 정확한 해로 접근시킬 수 있다. 야코비 반복법을 사용하여 연립방정식 <Eq. 3>을 푸는 경우 <Table 4>를 얻는다.계산은 모두 5 자리 째를 반올림했고 이 경우에는 반복을 6회 되풀이해서 참의 해 $x_1=1, x_2=-1, x_3=2$ 가 (5 자리 째로 반올림 함) 얻어진다.

Table 4. The Final Tabulation

	초 기 근 사	제 1 근 사	제 2 근 사	제 3 근 사	제 4 근 사	제 5 근 사	제 6 근 사
x_1	0	0.850	1.005	1.0025	1.0001	0.99997	1.0000
x_2	0	-1.3	-1.035	-0.9980	-0.99935	-0.9999	-1.0000
x_3	0	1.8	2.015	2.004	2.000	1.9999	2.0000

자코비 반복법이 반복 수가 많은 단점을 계량한 가우스-사이텔의 반복법 (Gauss- Seidel iteration) 또는 연차반복법(successive displacements) 등 많은 방법이 연구된 바 있다(김정수).

2. 자코비안(Jacobian) 행렬

모든 벡터는 열벡터를 의미하기로 한다. 또, $D_1, \cdots, D_n$ 과 같이 편미분기호, 즉 $D_k = \partial / \partial x_k$ 이다. 그리고 $F : R^n \to R^m$ 을 사상이라 하고 F를 좌표함수를 써서 표시한다. 즉, R^n 에서 R로 가는 함수 $f_1, \cdots, f_m$ 이 존재하여

$$F(X) = \begin{pmatrix} f_1(X) \\ f_2(X) \\ \text{M} \\ f_m(X) \end{pmatrix} = (f_1(X), \text{L} \ , f_m(X))^t$$

<Eq. 6>

또 $X = (x_1, \cdots, x_n)$ 로 표현된다. 지금 $f_1, \cdots, f_m$의 편도함수가 존재한다고 가정하고 이 편도함수들로 다음과 같이 행렬을 만들자.

$$\left(\frac{\partial f_i}{\partial x_j} \right) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \text{L} & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \text{L} & \frac{\partial f_2}{\partial x_n} \\ \text{M} & \text{M} & \text{M} & \text{M} \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \text{M} & \frac{\partial f_m}{\partial x_n} \end{pmatrix} = \begin{pmatrix} D_1 f_1(X) & \text{L} & D_n f_1(X) \\ \text{M} & \text{M} & \text{M} \\ D_1 f_m(X) & \text{L} & D_n f_m(X) \end{pmatrix}$$

<Eq. 7>

이 행렬을 F의 Jacobian 행렬이라 부르고 $J_F(X)$ 로 표시한다. 만일, $F: R^2 \to R^2$ 가 사상이고 F의 좌표함수가 f, g 라면, $F(x, y) = (f(x, y), g(x, y))^t$ 이고 F의 Jacobian 행렬은

$$J_F(x, y) = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix}$$

<Eq. 8>

가 된다.

3. 영상 복원에서의 자코비언 계산

<Figure 10>은 ET 영상복원에서 자코비안 함수가 단일 CPU를 위한 MATLAB 코드로 작성된 프로그램을 보이고 있다(Vauhkonen, 1998). 자코비언 계산에서 각 요소가 변수가 되며 각 요소는 벡터 rho로 표현되며 영상복원 루프의 진행에 따라 변화한다. Agrad 행렬은 요소의 저항값에 영향을 줄 수 있는 주변 요소들의 영향도를 갖고 있는 희소행렬로서 각 변수에 대한 편미분 방정식이 일차 연립방정식으로 변환된 것이다. 이 Agrad 행렬은 영상복원 과정 중 FEM을 변환하는 과정에서 구해진다. NNode는 노드의 개수를 저장하는 변수로서 여기서의 노드란 <Figure. 1(b)>에서 삼각형 요소들의 꼭지점을 의미한다. U0는 주입된 신호를 각 전극에서 측정한 값이고 U는 현재 추정된 rho 값을 기반으로 각 전극에서 측정될 것으로 예상되는 값이다. U는 영상 복원 과정중 Forward solver에 의해 계산된다. J는 계산된 자코비언 행렬의 결과가 저장되는 자료구조이다.


```
J = zeros(992, 776);
for i=1:1:776
    J(:,i) = U0.'*1/rho(i)^2 *
             reshape(Agrad(:,i), NNode, NNode) * U;
end
```

Figure 10. MATLAB code for Jacobian

본 논문의 영상 복원 모델은 속도계산을 목적으로 하므로 각 행렬의 차원이 중요한데 요소의 수는 992로 설정되어 있으므로 rho는 992 개의 원소를 갖는다. 요소의 크기는 <Figure 1. (b)> 에서와 같이 전극 주변, 즉 물체의 가장자리로 갈수록 작아지며 요구되는 복원 영상의 해상도와 전극의 수, 주입되는 전극의 신호 개수에 의해 결정된다. 요소의 수가 다른 행렬의 차원에 크게 영향을 주게 되는데 J는 992 * 776, U0와 U는 각각 1681 * 31, 1681 * 32이다. 주어진 메쉬 구조상 NNode는 1681의 값을 갖는다. J의 원소수는 전극의 수 32, 주입된 신호의 패턴 31의 곱, 즉 992에 대해 각 요소의 미분값을 갖기 위해 992*776개의 원소를 갖는다. U0는 주입

된 패턴에 대해 각 노드들에서 계산되는 값이고 U는 노드의 값에 대해 각 전극에서 측정될 것으로 예상되는 값이다. 이 과정에서 요소의 수가 아닌 노드의 수가 행렬의 원소수를 결정하는 이유는 주어진 영상복원 과정에서 Forward Solver나 FEM 방식이 노드의 전류값에 기반하여 U, U0를 계산하기 때문이다. 또 Agrad 행렬은 극심한 정도의 희소행렬로서 2825761 * 776 행렬인데 각 요소를 위한 방정식은 24개에 불과하다.

<Figure 10>에서 보이는 변수들은 대부분 영상 복원 루프 진행에 영향을 받지 않는다. 즉 메쉬 구조는 루프 진행에도 불변하므로 분산 계산을 수행할 경우 각 컴퓨터들로 하여금 사전에 이 자료구조를 갖고 있도록 하면 된다. 또 Agrad는 이미 측정된 데이터들로서 구축된 연립방정식이고 U0는 측정된 신호에 의해 결정되므로 역시 각 컴퓨터들이 사전에 전달받을 수 있다. 단 U는 Forward solver에 의해 값이 변경되는데 이 Forward solver 프로그램은 수행시간이 미약하므로 각 노드에서 각각 수행되어도 전체적인 성능에 큰 영향을 주지 않는다. 단 rho 벡터는 매 복원 루프마다 계속 갱신되고 다음 계산에 영향을 주기 때문에 자코비언 계산을 하는데 있어서 데이터 복사의 대상이 된다. <Figure 10>에서 자코비언 계산은 요소의 수, 776 번의 루프를 수행하면서 자코비언 행렬의 한 열을 계산하는데 한 루프는 행렬과 벡터의 곱으로 구현되어 있다. 각 루프간에는 데이터 의존성이 없기 때문에 독립적으로 분산되어 수행되어도 무방하며 클러스터에 포함된 노드의 수와 성능에 따라 루프를 적절히 분배하여 수행시킬 수 있다.

4. 분산 자코비안 계산 기법

분산 계산은 하나의 마스터 노드가 작업을 분리하여 몇 개의 슬레이브 노드에 나누어주고 각각 할당된 작업을 수행한 후 부분 결과를 결합하는 과정으로 구성되는데 네트워크를 통한 자료구조의 전달이 포함된다. 분산 계산은 C언어로 구현되었다. 자코비안 계산은 복원 루프의 한 부분으로 구성되는데 복원 루프가 반복됨에 따라 변경되는 것은 rho이다. 따라서 분산 계산을 함에 있어서 분산된 노드는 rho

를 제외한 다른 행렬은 오프라인 시에 미리 값을 알고 있을 수 있으며 슬레이브 노드들은 rho만 새로이 수신하여 자코비언의 부분 계산을 수행한다. 즉, rho를 제외한 다른 행렬들은 영상복원 과정 시작 이전에 사전에 네트워크 혹은 디스크 연산을 통해 변수로 생성되어 있다고 가정한다. 실제 구현에서는 디스크에서 데이터를 적재한다. 이후 슬레이브 노드들은 결과를 마스터에게 전달하게 되며 마스터는 이를 결합하여 J 행렬을 생성한다. 이를 MPI 라이브러리를 기반으로 구성하기 위해서는 마스터 코드와 슬레이브 코드로 구분하여야 하는데 MPI_rank 함수의 리턴 값에 의해 구분이 가능하다.

행렬을 전달하기 위해서는 행렬을 표현하는 방식과 전달하는 함수를 정확하게 연결시켜야 한다. 일반적인 산술계산 라이브러리에서는 행렬을 저장하는데 있어서 구조체를 기반으로 행렬의 이름, 차원, 데이터 블록 포인터 등 행렬의 특성을 기술하는 한편 실제 데이터 블록에 열 우선으로 각 원소들을 저장한다. 각 원소는 C 언어에서 double 형의 타입을 가지므로 한 원소가 8 바이트로 이루어진다(김철민, 이정훈, 2001). 이 행렬의 전송은 MPI 함수를 이용하는데 송신자는 행렬 기술자(descriptor)에 의해 전송할 행렬의 바이트 수를 계산하는 한편 MPI 함수의 인자로 데이터 블록의 포인터를 넘겨 주어야 한다. 행렬의 일부를 전송하려 하는 경우에는 열 우선의 데이터 저장 방식을 고려하여 전송될 원소들의 시작 위치를 계산하여야 함은 물론 원소의 수에 기반하여 전송될 바이트 수를 계산하여야 한다. 수신자 측에서는 수신하려는 행렬을 위한 공간을 미리 할당한 후 그 위치에 네트워크를 통해 데이터를 저장하여야 한다. 자코비언 계산을 위한 행렬들은 루프의 진행에도 변경되지 않고 고정된 값이므로 수신자는 미리 알 수 있으며 이를 이용하여 산술 계산 라이브러리를 호출하여 필요한 기억공간을 할당받을 수 있다. 할당을 받으면 행렬의 기술자와 아울러 데이터 블록을 위한 공간이 생성되어 리턴되는데 MPI_receive 함수의 인자로 이 차원과 데이터 블록에 대한 포인터를 주면 수신이 가능하다.

rho가 각 노드에 방송된 후 각 노드는 할당된 루프를 수행하는데 하나의 루프는 1회의 스칼라 곱하기, 2회의 2차원 행렬 곱하기 및 행렬의 reshape 등으로 구성된다. 이를 수행하기 위해서는 산술계산 라이브러리에서 제공하는 함수들을 이용하는 것이 바람직하다. MATLAB, LAPACK 등의 산술계산 라이브러리들은 고속의

행렬 계산 기능을 갖고 있으며 C 언어를 위한 API(Application Program Interface)를 제공한다(Mathworks). 계산 결과를 또다른 행렬로 반환하여 다음의 연산에 쉽게 이용될 수 있도록 하므로 한번 계산에 사용되고 추후 사용되지 않는 행렬은 free 함수에 의해 히프 메모리로 반환되어야 한다. 반면 스칼라 곱하기는 계산량이 적기 때문에 산술계산 라이브러리 호출의 오버헤드를 피하기 위해 여러 행렬 중 가장 원소의 수가 작은 행렬에 대해 원소별로 곱을 수행하는 것이 효율적이다.

부분 계산을 수행한 후 결과를 마스터에게 전달하는데 J의 원소수가 많기 때문에 네트워크 연산에 많은 시간이 소요된다. 더욱이 3 개 이상의 노드에서 분할 수행된 후 마스터에 보고되는 경우 동시 네트워크 접근에 의한 충돌로 전송시간은 예측할 수 없이 증가할 가능성도 내포하고 있다. 마스터는 사전에 J 행렬을 위한 기억장소를 할당한 후 각 노드가 계산한 부분 결과를 결합하여야 하는데 각 마스터는 자신의 rank를 갖고 있으므로 마스터는 슬레이브의 rank에 의해 해당 부분 결과를 저장하는 J 행렬 내에서의 위치를 계산한다. 이 과정에서 마스터가 각 슬레이브로부터 순차적으로 수신하므로 계산이 먼저 종료한 슬레이브의 데이터가 먼저 수신되는 것은 아니다.

MPI 함수를 이용하여 행렬을 교환하는데 있어서 MPI_send 함수를 사용하였는데 노드의 수가 늘어난다면 멀티캐스트 함수를 사용하는 방안이 바람직할 것으로 판단된다. 또 이와 아울러 MPI 라이브러리에서 제공하는 비동기 송수신 연산을 이용하여 구현한다면 순차적 수신에 비해 보다 효율적으로 통신 연산이 수행될 수 있다.

V. 성능평가

본 장에서는 구축된 클러스터와 구현된 분산 자코비언 계산 기법의 성능을 평가하기 위해 프로그램을 클러스터에서 수행한 후 그 수행시간을 측정하였다. <Table 3>에서 보는 바와 같이 클러스터를 이루는 3 개의 노드 중 하나는 그 성능이 많이 떨어지므로 이 노드가 계산에 참여하면 전체적인 성능이 크게 저하된다. 따라서 대부분의 실험은 2 대의 노드를 이용하여 수행되었으며 좋은 성능을 갖는 PC가 클러스터에 도입될 경우 더 나은 성능을 기할 수 있다. 수행된 실험은 3 개의 결과를 보이는데 먼저 통신 오버헤드에 대한 실험으로서 분산 알고리즘을 수행하는데 있어서는 통신 오버헤드가 전체적인 성능에 큰 영향을 차지한다. 다음에 분산 계산의 전형적인 테스트베드 프로그램인 행렬의 곱하기 프로그램을 구현하여 수행시킨 후 그 수행시간을 측정하였다. 마지막으로 구현된 분산 자코비언 계산 프로그램을 2 개의 노드에서 수행시킨 결과를 보이는데 각 노드에 할당된 루프의 양을 변경시켜가며 수행시간을 측정하였다.

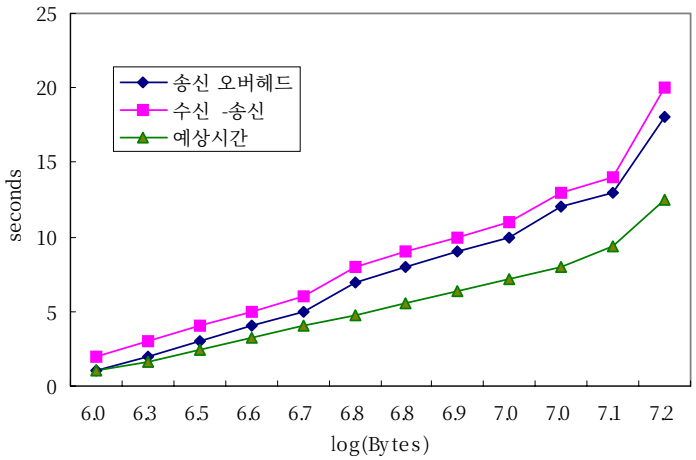


Figure 11. Communication overhead

<Figure 11>은 전송할 바이트에 따르는 전송시간을 보이고 있는데 이 그래프는 실제적으로 MPI 라이브러리의 통신 성능을 보이고 있다. 실험에서 하나의 송신자에서 수신자까지 단방향으로 전송할 바이트 수를 변화시켜가며 이에 따르는 수신시간을 측정하였다. 단방향이므로 CSMA/CD 프로토콜에 따르는 충돌은 거의 발생하지 않는데 충돌이 발생하는 경우는 MPI 라이브러리 수행 중 확인과 같은 메시지가 수신자측에서 전송되기 때문이다. 클러스터 내에서의 통신이므로 다른 트래픽은 존재하지 않는다. 그림에서 맨 밑의 곡선은 소프트웨어 오버헤드를 제외하고 순수한 네트워크 전송시간을 계산한 것으로서 10 Mbps 대역폭을 기준으로 전송 비트를 단순하게 나눈 값을 표시한 것으로서 MPI 라이브러리 오버헤드의 효율성을 보이는 기준이 된다. 가운데 그래프는 송신자 측에서 측정한 통신이 완료될 때까지의 시간으로 송신자와 수신자간에 통신 동기화 간격에 대한 정보를 제시한다. 가장 위의 곡선은 송신자 노드가 MPI 함수를 호출한 후 수신자 측에서 수신의 완료될 때까지의 시간을 측정한 것인데 물론 이 측정시간에는 송신자와 수신자의 시계 동기화 오류에 기인한 오류값이 포함되어 있으나 전체적인 측정시간이 크므로 이 오류의 영향은 미약하다. <Figure 11>에서 보는 바와 같이 단방향으로 데이터의 이동이 수행될 경우 순수 네트워크 전송시간 대 전체 시간(소프트웨어 오버헤드 포함)은 1.8 배 이내로서 소프트웨어 오버헤드가 상대적으로 낮음을 알 수 있다. 분산 자코비언 계산에 있어서 10^6 바이트의 전송이 수반되는데 이의 오버헤드는 7~8 초 정도이다.

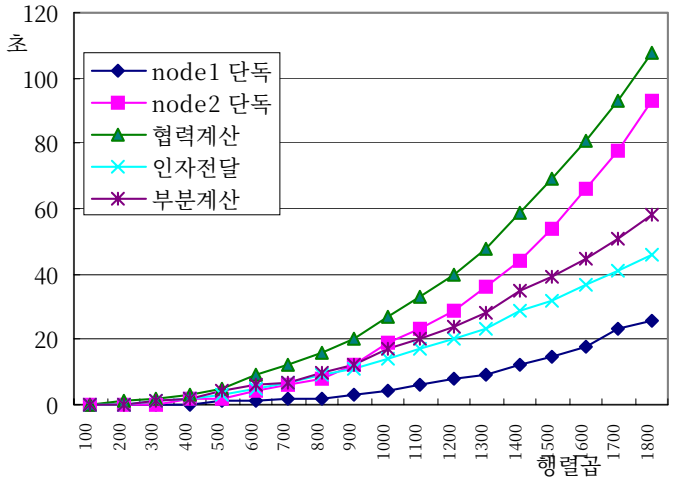


Figure 12. Analysis of matrix double

<Figure 12>는 두 개의 2차원 행렬 곱하기를 수행시킨 결과이다. x축은 원소수를 나타내는데 100 이면 두 개의 100*100 행렬의 곱을 수행한 것을 의미하며 y축은 이에 따르는 수행시간을 측정한 결과이다. A*B를 수행하기 위해 마스터는 B 행렬을 B1, B2 등 2 개의 서브 행렬로 분할한 후 슬레이브에게 A와 B2를 전달하며 자신은 A*B1을 수행한다. 슬레이브가 계산을 끝내면 부분 결과를 마스터에게 전달한다. 이때 만약 두 개의 n*n 행렬 곱을 수행한다면 통신에 필요한 바이트 수는 2 * n * n * sizeof(double)이 된다. 각 노드에서의 부분 계산은 산술계산 라이브러리가 제공하는 곱하기 라이브러리, MATLAB에서는 mlfMtimes() 함수를 사용한다. 행렬의 분할은 포인터 변수의 조작으로 간단하게 이루어질 수 있으며 결과의 결합은 사전에 할당된 메모리 영역에 슬레이브로부터의 결과를 복사하면 된다. 그림에서 보는 바와 같이 성능이 우수한 노드 1에서 단독으로 수행한 것이 가장 효율적인데 이는 행렬의 곱하기는 많은 네트워크 연산을 포함하므로 네트워크 전송시간이 전체적인 성능을 많이 저하시키기 때문으로 분석된다. 협력 계산이 노드 2에서 단독으로 수행하는 것보다도 수행시간이 오래 걸리는데 이를 개선하기 위해서는 효율적인 부하 분배 알고리즘이 도입되어야 할 것이다. 협력 계산은 네트워크 전송과

부분계산 및 부가적인 오버헤드로 구성되는데 네트워크 전송시간이 43 % 까지 차지한다.

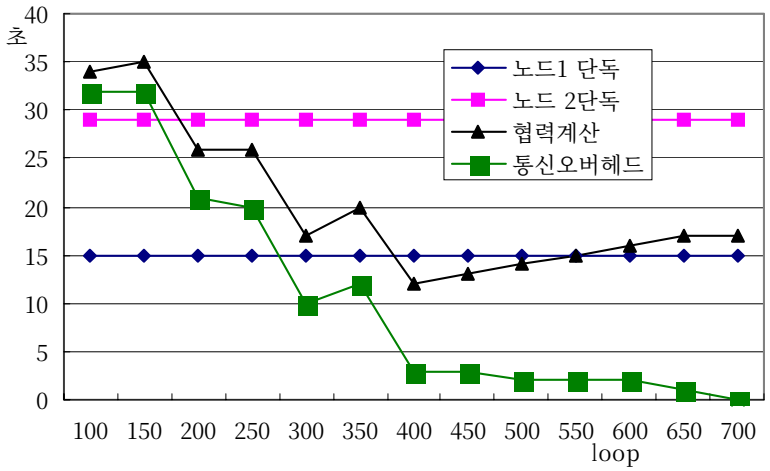


Figure 13. Analysis of Jacobian

<Figure 13>은 본 논문에서 제시한 분산 계산 기법에 의해 자코비언을 수행한 결과이다. 776 번의 루프로 계산됨에 있어 각 노드에게 분배되는 부하가 전체적인 성능에 영향을 주게 된다. 노드 1에게 많은 계산을 할당하면 좋은 성능을 기할 수 있는데 그림에서 보는 바와 같이 400 개의 루프를 노드 1에게 할당할 때 가장 빠른 수행시간을 보인다. 노드 1에 많은 부하가 할당될수록 통신 오버헤드는 감소하면 또 전체적인 수행시간이 감소함을 알 수 있다. 노드 2에 많은 부하가 할당되는 경우는 통신오버헤드 뿐만 아니라 노드 2가 부분 계산을 수행하는데 병목지점으로 동작하므로 전체적인 성능이 저하된다. 노드 1과 노드 2는 그림에서 보는 바와 같이 거의 2 배 정도의 성능 차이를 보이고 있다. 역시 네트워크 전송시간이 전체 계산에 있어서 많은 부분을 차지하므로 네트워크 장비의 성능이 개선되고 고속화된다면 분산 계산의 효율성을 기할 수 있다. 이와 아울러 유사한 성능을 갖는 노드들로 클러스터를 구성한다면 보다 효율적인 부하 할당을 할 수 있어서 성능의 개선을 기할 수 있다. 또 데이터의 독립성이 보장되므로 노드의 수가 늘어난다면 네트워크

전송에 있어서의 오버헤드는 개선하지 못한다하더라도 부분 계산에 있어서의 수행 시간을 개선할 수 있다.

VI. 결론

본 논문에서는 막대한 계산 시간을 수반하는 전기 단층 촬영에 의한 영상 복원에 있어서 속도를 개선하기 위하여 LINUX 운영체제, MPI, 산술계산 라이브러리에 기반한 클러스터를 구축하였다. 클러스터는 각각 다른 성능을 갖는 CPU와 메모리를 갖고 있다. 이를 바탕으로 영상 복원 과정중 가장 많은 시간을 차지하는 자코비언 계산에 대해 그 특성과 자료구조를 분석하고 루프의 독립성을 바탕으로 마스터-슬레이브 구조에 기반한 분산 계산 기법을 제시하였다. 제시된 계산 기법은 영상 복원 수행 이전에 각 노드가 정적인 데이터를 적재하고 마스터가 요소들의 현재 값을 슬레이브에게 넘겨주면 각 노드들은 병렬적으로 계산을 수행한다. 이후 부분 결과들은 다시 마스터에게 전송되어 결합됨으로써 최종적인 자코비언 행렬이 계산된다. 이 과정에서 데이터의 전달을 위해 산술계산 라이브러리에서 행렬을 표현하는 방식에 따라 실제 원소들의 위치를 파악한 후 MPI 라이브러리를 호출하는데 MPI는 클러스터 컴퓨팅의 메시지 교환에 있어서 표준으로 제정되어 있다. 반면 데이터가 분할된 후 각 노드들은 설치되어 있는 산술계산 라이브러리를 호출함으로써 효율적으로 수행되며 새로운 산술계산 라이브러리로 변경이 가능하다.

프로그램의 구현은 C 언어를 사용하였으며 C 프로그램에서 MPI 라이브러리와 산술계산 라이브러리를 호출한다. 이 프로그램은 마스터와 슬레이브 코드를 모두 포함하고 있으며 수행이전에 각 노드에 적재되어 있다. 새로운 노드가 클러스터에 추가되는 경우 이 프로그램만 사전에 적재하고 있으면 되고 이 프로그램은 노드의 수에 상관없이 동작하므로 확장성이 보장된다. 또 프로그램은 추후의 부하 균배 방식의 도입을 위해 커맨드 라인 상에서 루프를 마스터와 슬레이브에 분할할 수 있도록 작성되어 있다. 이 프로그램을 비교적 성능이 좋은 두 대의 노드에서 수행시킨 결과 마스터와 슬레이브의 부하 비율이 400:376 정도에서 가장 좋은 성능을 보이고 있으며 마스터에 할당된 부하의 비율이 높은 경우 단일 CPU에서 수행한 결과보다 현재의 구조상으로도 좋은 성능을 보이고 있다. 그러나 현재 성능 차이가 현저한 3

대의 노드로 클러스터가 구성되어 있으므로 획기적인 성능향상을 기할 수는 없다. 또 자코비언 계산은 네트워크를 통한 전송시간이 많은 부분을 차지하기 때문에 성능 개선에 장애가 된다.

결국, 본 논문에서 제시된 자코비언의 분산 계산 기법은 클러스터 상에서 수행되어 단일 CPU 수행에 비해 18 % 정도의 개선을 달성할 수 있었으며 다음의 요소들이 추가적으로 보완되거나 연구된다면 더많은 속도의 개선을 기대할 수 있다. 첫째, 네트워크의 성능 개선으로서 현재 10 Mbps를 고속의 네트워크로 대치한다면 전송시간을 줄일 수 있으며 3 대 이상의 노드들이 협력 계산하는 경우를 위해 하드웨어 혹은 소프트웨어 프로토콜 면에서 멀티캐스트 기능이 적용되어야 한다. 둘째, 노드의 성능 개선으로 산술계산 라이브러리의 수행속도는 각 노드의 CPU와 메모리 양에 의해 좌우된다. 이와 아울러 이중 CPU 혹은 DSP(Digital Signal Processor) 등과 같은 고속의 계산 기능을 갖는 하드웨어를 설치한 노드의 추가는 전체적인 성능을 크게 개선할 것으로 예상된다. 또 성능차이가 있는 노드들로 클러스터를 구성하기 보다는 성능이 동일한 PC들을 클러스터에 도입하는 것이 바람직하다. 마지막으로, 보다 많은 노드들이 클러스터에 추가되고 다양한 기능을 갖는 경우에 효율적인 부하 균배 알고리즘의 도입이 성능 개선에 기여할 것으로 기대된다.

VII. 참고문헌

Almasi, G., A. Gottlieb, 1998, Highly Parallel Computing, The Benjamin/Cummings Publishing Company.

Baker, M., 2000. Cluster Computing White Paper: Status - *Final Release Version 2.0*.

Bo-Sung.Lee., 2000. 리눅스 클러스터링 최신 기술동향, 『KORDIC 초고속망 워크샵.』

Bozic S., Digital and Kalman Filtering Edward Arnold. pp.81~140.

Bo-Sung.Lee., 2000. Linux Super Computer 공동개발 프로젝트, 『KORDIC 초고속망 워크샵.』

Bung, R., D. L. Eager, G. M. Oster, and C. L. Williamson. 1999. "Achieving Load Balance and Effective Caching in Clustered Web Servers". In *Proc. of 4th International Web Caching Workshop*, San Diego, CA. pp.159~169.

Buyya, R., 1999. High Performance Cluster Computing: Programming and Applications, Vol. 2, Prentice Hall.

Cheney, M., et. al., 1999. Electrical impedance tomography, *SIAM Review*, No, 41, pp.85~101.

Greg Burns, Raja Daoud, James Vaigl. LAM: An Open Cluster Environment for MPI, *Ohio Supercomputer Center Columbus, Ohio.*

Greg Burns, Raja Baoud. Robust MPI Message Delivery with Guaranteed Resources, *Ohio Supercomputer Center Columbus, Ohio.*

Jordan, A., R. Bycul. 2001. The parallel algorithm of conjugate gradient method, *International Workshop on Cluster Computing(LNCS2326)*. pp.156~165.

Libertone, D., 2000. Windows 2000 Cluster Server Guide Book, A Guide to

Creating and Managing a Cluster, 2nd Ed., Microsoft Technology Series.

Lusk, E., et. al., 1997. Message Passing Interface Forum, MPI-*2: Extensions to the Message-Passing Interface* .

Mathworks, MATLAB C Math Library : *User's Guide Version 2*

Vauhkonen, M., at. al, 1998. A Kalman filter approach to track fast impedance changes in eletrical impedance tomography, *IEEE Trans. on Biomedical Engineering.* pp.486~493.

Wadleigh, K., I. 2000. Crawford, Software Optimization for High Performance Computing, Prentice Hall.

김규철, 나연목. 『알기쉬운 수치해석』 제 2판 시그마프레스. pp.348~355.

김정수, 박을룡, 이일해, 윤옥경, 고영소, 김성기. 『미적분학』 이우출판사. pp.334~337.

김철민, 이정훈. 2001. 이중 CPU PC에서 병렬 계산을 위한 MATLAB 행렬 연산 라이브러리의 구현 및 성능 측정, 『정보과학회 추계학술대회(Ⅲ).』 pp.871~873.

권재식, 2000. 『21세기의 과제, 리눅스 클러스터링.』

문태준, 『리눅스 시스템 모리터링 시스템 최적화.』

신순철, 『부하분산 클러스터 제작과 Fail Over.』

이장우, 『선형대수의 입문』, 京文社. pp.462~470.

이정훈, 손수방. 2002. ET 영상복원에서 클러스터 컴퓨팅에 의한 자코비언 계산의 속도 향상 기법, 『정보과학회 추계학술대회(Ⅰ)』, pp.343~345.

이형구, 『리눅스 클러스터링 & 로드발런싱; 강좌 버전 1.0.』

VIII. 소스

```
#include "mpi.h"
#include <string.h>
#include "/usr/local/matlab/extern/include/matlab.h"
#include <sys/time.h>
#include <unistd.h>

CalcDim(mxArray *arr)
{
        mxArray *dim;
        double *value;
        int     intval;
        dim = mlfSize(NULL, arr, NULL);
        value = mxGetPr(dim);
        intval = ((int) (*(value+ 1)));
        mxDestroyArray(dim);
        return(intval);
}

void CopyToJ(mxArray *J, mxArray *jj, int i)
{
        double *src, *dest;
        int     m,n;
        int     bytes;
        m = mxGetM(jj);
        n = mxGetN(jj);

        dest = mxGetPr(J);
        dest += m*n*i;
        src = mxGetPr(jj);
        bytes = m*n* sizeof(double);
        memcpy(dest, src, bytes);

   return;
}



int main(int argc, char **argv)
{
   int rank, size,result;
   int  tmps;
   int  Jtrans;
   MPI_Status    status;
        mxArray *J=NULL, *Node,*Elem,*Agrad, *U, *U0, *rho =NULL;
        mxArray *Utrans, *aTrac;
        mxArray *idx1, *idx2, *tmp, *tmp1, *tmp2, *tmp3, *tmp4, *tmp5;
        mxArray *fname;
        int nNode, nElem, aSize, i;
        double *rhoData;
        double factor;
        struct timeval h1, h2;
        double *Jptr;

        Jtrans  = atoi(argv[1]);
        fname = mxCreateString("./haha.mat");
```

```
        mlfLoad(fname, "Node2", &Node, "Element2", &Elem, "Agrad1",
&Agrad,
                        "Ucur", &U, "Umeas", &U0, "rho", &rho, NULL);
        mxDestroyArray(fname);

        fname = mxCreateString("./Jmat.mat");
        mlfLoad(fname, "J", &J, NULL);
        mxDestroyArray(fname);

        nNode = CalcDim(Node);
        nElem = CalcDim(Elem);
        aSize = CalcDim(Agrad);

        rhoData = mxGetPr(rho);
        Utrans = mlfTranspose(U0);

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if(rank ==0) {
        sleep(2);
        gettimeofday(&h1, NULL);

        MPI_Send((char *) rhoData, 776* sizeof(double),
        MPI_UNSIGNED_CHAR, 1, 123, MPI_COMM_WORLD);

        gettimeofday(&h2, NULL);
        printf("%5d  %5ld ", Jtrans, h2.tv_sec - h1.tv_sec);

        for (i=0; i< Jtrans; i++) {
                factor = *(rhoData + i);
                factor = factor * factor;
                factor = 1.0 / factor;
                idx1 = mlfScalar(factor);

                tmp2 = mlfMtimes(Utrans, idx1);
                mxDestroyArray(idx1);

                idx1 = mlfScalar((i+1)* 1.0);
                aTrac = mlfIndexRef(Agrad, "(?,?)",
mlfCreateColonIndex(),idx1);
                mxDestroyArray(idx1);

                idx1 = mlfScalar(nNode*1.0);
                idx2 = mlfScalar(nNode*1.0);
                tmp3 = mlfReshape(aTrac, idx1, idx2, NULL);
                mxDestroyArray(idx1);
                mxDestroyArray(idx2);

                tmp4 = mlfMtimes(tmp2, tmp3);

                mxDestroyArray(tmp2);
                mxDestroyArray(tmp3);
                mxDestroyArray(aTrac);
                tmp5 = mlfMtimes(tmp4, U);

                mxDestroyArray(tmp4);
                CopyToJ(J, tmp5, i);
                mxDestroyArray(tmp5);
```

```
        }

        gettimeofday(&h2, NULL);
        printf(" %5ld ", h2.tv_sec - h1.tv_sec);

        Jptr = mxGetPr(J);
        MPI_Recv((char *) Jptr, 992 * (776 - Jtrans) * sizeof(double),
            MPI_UNSIGNED_CHAR, 1, 123, MPI_COMM_WORLD, &status);

        gettimeofday(&h2, NULL);
        printf(" %ld₩n", h2.tv_sec - h1.tv_sec);

    }
   else {

         MPI_Recv((char *) rhoData, 776 * sizeof(double),
           MPI_UNSIGNED_CHAR, 0, 123, MPI_COMM_WORLD, &status);

        for (i=0; i< 776 - Jtrans; i++) {
                factor = *(rhoData + i);
                factor = factor * factor;
                factor = 1.0 / factor;
                idx1 = mlfScalar(factor);

                tmp2 = mlfMtimes(Utrans, idx1);

                mxDestroyArray(idx1);

                idx1 = mlfScalar((i+1)* 1.0);
                aTrac = mlfIndexRef(Agrad, "(?,?)",
mlfCreateColonIndex(),idx1);
                mxDestroyArray(idx1);

                idx1 = mlfScalar(nNode*1.0);
                idx2 = mlfScalar(nNode*1.0);
                tmp3 = mlfReshape(aTrac, idx1, idx2, NULL);
                mxDestroyArray(idx1);
                mxDestroyArray(idx2);

                tmp4 = mlfMtimes(tmp2, tmp3);
                 mxDestroyArray(tmp2);
        }
         Jptr = mxGetPr(J);
        MPI_Send((char *) Jptr , 992* (776 - Jtrans)* sizeof(double),
        MPI_UNSIGNED_CHAR, 0, 123, MPI_COMM_WORLD);
    }
    MPI_Finalize();
        mxDestroyArray(Node);
        mxDestroyArray(Elem);
        mxDestroyArray(Agrad);
        mxDestroyArray(U);
        mxDestroyArray(U0);
        mxDestroyArray(rho);
        mxDestroyArray(J);
   return 0;
}
```