ISBN 978-89-5884-984-1 98560

# 클러스터 시스템 환경에서의 병렬프로그램 성능 측정

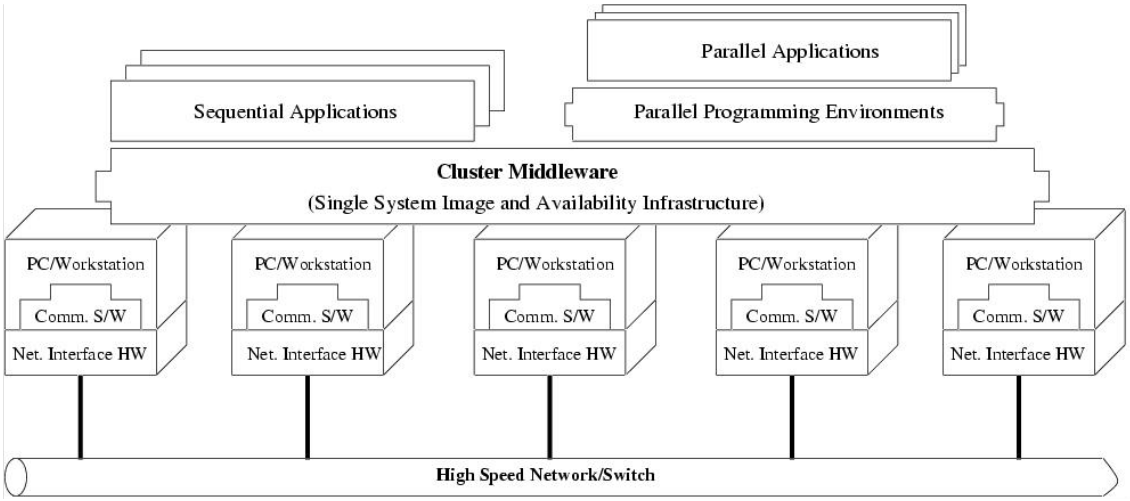
일 자: 2007년 12월 7일
부 서: 클러스터개발팀
제출자: 차광호

KiSTi www.kisti.re.kr Korea Institute of Science and Technology Information

# 목 차

# 1. 개 요

클러스터 시스템이란 다수의 PC 또는 워크스테이션을 고성능의 네트웍를 이용하여 연결, 하나의 시스템으로서 함께 작동하는 컴퓨터를 말한다. IBM의 Gregory F.Pfister는 그의 저서 “In Search of Clusters”에서 클러스터를 “A type of parallel or distributed system that consists of a collection of interconnected whole computers, and is used as a single, unified computing resource.”로 정의하고 있다. 여기서 ‘whole computer’는 1개 이상의 프로세서, 적당량의 메모리, 입출력 장치 및 운영체제를 독자적으로 가지있어 단독으로 사용될수 있는 컴퓨터를 의미한다. 다음 그림은 클러스터 시스템의 구성을 보여주고 있다.




클러스터를 사용하는 이유는 무엇보다도 가격대 성능비가 우수하기 때문이다. 클러스터는 대량 생산되는(COTS:commercially off-the-shelf)부품, 즉 성능은 우수하나 가격이 저렴한 PC, 네트워크장비 및 무료 소프트웨어 등을 최대한 사용함으로서 가격을 기존의 동급 병렬컴퓨터에 비해서 십분의 일 이하로 낮출 수 있다.

최근 클러스터가 관심이 집중되는 이유는 가격대비 성능이 우수한 부품이 대량 생산되고 있으며 특히 고속 네트워크 장비의 개발과 양산으로 인해 고성능 클러스터의 제작이 가능하다고 보기 때문이다. 클러스터 시스템이 원하는 성능을 내기 위해서는 다수의 PC를 연결하는 연결망이 충분한 성능을 내주어야 한다. 기존의 슈퍼

컴퓨터는 연결망에 있어서 독자적인 디자인으로 구성되어 프로세서간 통신이 매우 고속으로 이루어지는데 반하여, 과거의 PC간 통신수단들은 Ethernet(10~100Mbps)정도의 속도를 넘어설 수 없었다. 그러나 Gigabit Ethernet을 시작으로, Myrinet, QsNet, Infiniband등과 같은 고속 근거리 네트워크의 등장으로 속도 문제에 많은 개선을 보게되어, 성능면에서도 기존의 슈퍼컴퓨터와 동일한 수준의 시스템으로 자리매김하고 있다. 이는 매년 2번씩 전세계 슈퍼컴퓨터의 성능을 취합 발표하는 TOP500리스트의 상위권에 상당수의 클러스터 시스템이 랭크되고 있슴에서도 확인할 수 있다.

이와 같은 하드웨어적인 발전뿐 아니라, 소프트웨어적인 요소도 클러스터 시스템에 중요한 영향을 미치고있는데, 클러스터 시스템을 고성능, 고가용성 솔루션으로 사용할수 있게끔하는 표준화된 소프트웨어 환경의 보급을 예로 들수 있다. 즉, 안정적으로 제공되고있는 프리웨어인 리눅스와, MPI와 같은 표준화된 병렬 프로그래밍 환경을 생각할 수 있다. MPI(Message Passing Interface)의 경우, 업계 표준으로 자리잡고 폭넓게 사용되는데 쉬운 인터페이스와 적정한 성능을 보여 줌으로서 병렬프로그램도 쉽게 개발되며 또한 다른 컴퓨터에서의 호환성이 보장되서 쉽게 이식될 수 있다는 장점이 있다. 1992년부터 시작된 MPI에 대한 연구는 1994년 5월 MPI 1.0이 완성된 이후에도 지속적으로 발전하고 있으며, MPI 1.2의 경우 전체 125개의 API 중에서 기본적인 6개의 API만을 사용하여서도 충분히 병렬프로그램을 작성할 수 있어 개발이 용이해졌다.

이와 같은 배경으로 현재의 클러스터 시스템은 다양한 하드웨어를 바탕으로 다양한 운영체제와 병렬 프로그래밍 환경으로 구성될수 있다. 이에 본 보고서에서는 다양한 클러스터 시스템을 대상으로 기초적인 병렬프로그램들을 테스트하였다. 대표적인 병렬 프로그래밍 환경인 MPI환경을 위주로 테스트하였으며, 쓰레드 기반 병렬 프로그래밍 환경이 OpenMP도 함께 테스트하였다.

# 2. Matrix Multiplication

## 2.1. 문제 개요

성능 비교를 위하여 두가지 행렬 곱셈 알고리즘을 구현하고 클러스터 시스템에서 실행하였다. 첫번째 프로그램은 각 행렬을 행단위로 분할하여 프로세스 노드에 할당하는 Row-wise distribution 방식을 취하였으며, 두번째 프로그램은 각 행렬을 정사각형의 블록 형태로 분할하는 Checkerboard distribution방식으로 Fox's Algorithms을 사용하였다.

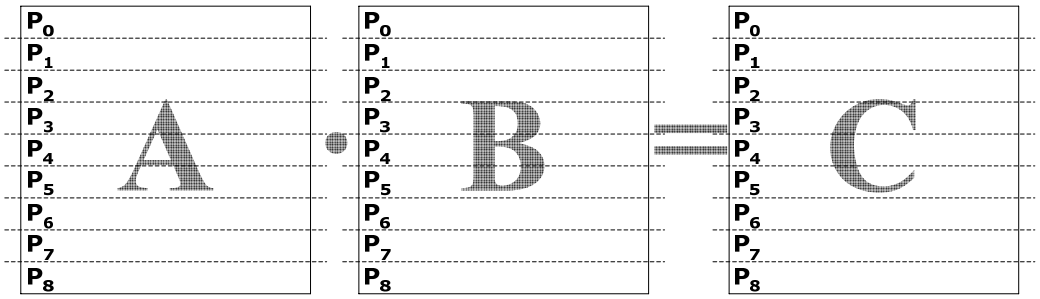



그림1. Row-wise distribution

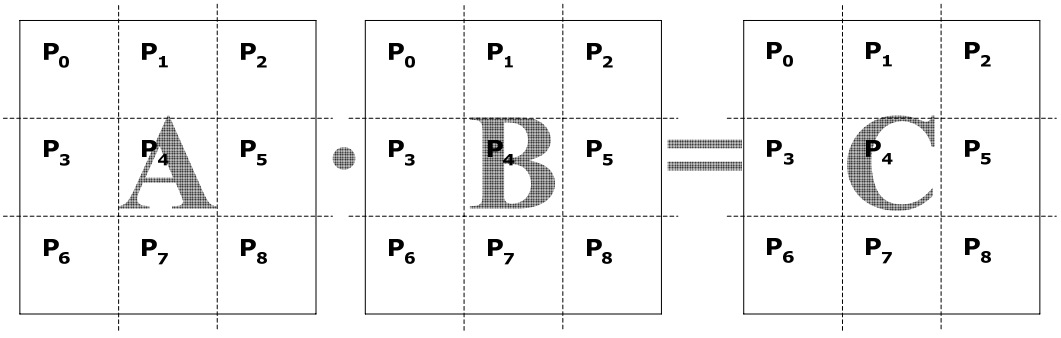


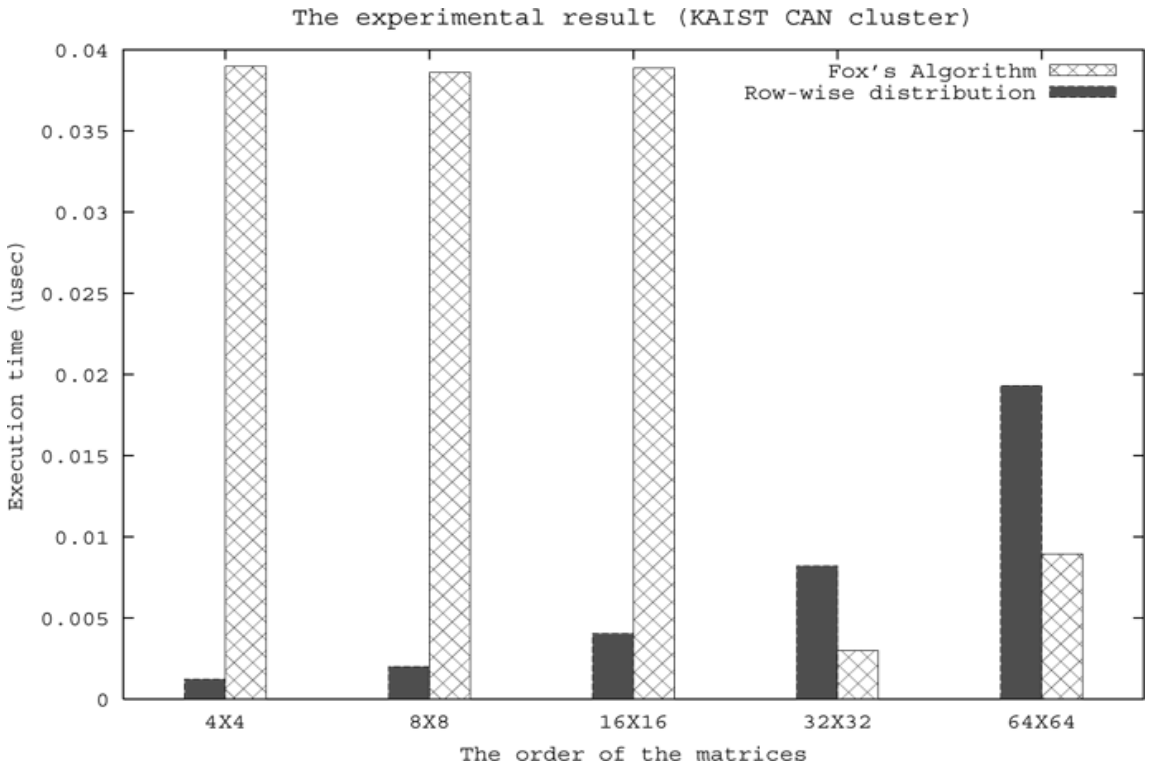

그림2. Checkerboard distribution

수행 시간은 두 행렬이 0번 프로세스 노드로 읽혀진 이후 시점부터, 계산이 완료되어 0번 프로세스 노드가 결과 행렬을 파일에 기록하기 바로 전까지 측정하였다.

## 2.2. 성능 측정

### A. KAIST CAN Cluster

다음은 Intel Pentium IV기반 클러스터 시스템인 CAN 클러스터 시스템에서 4개의 계산 노드를 이용한 성능 측정 결과이다.



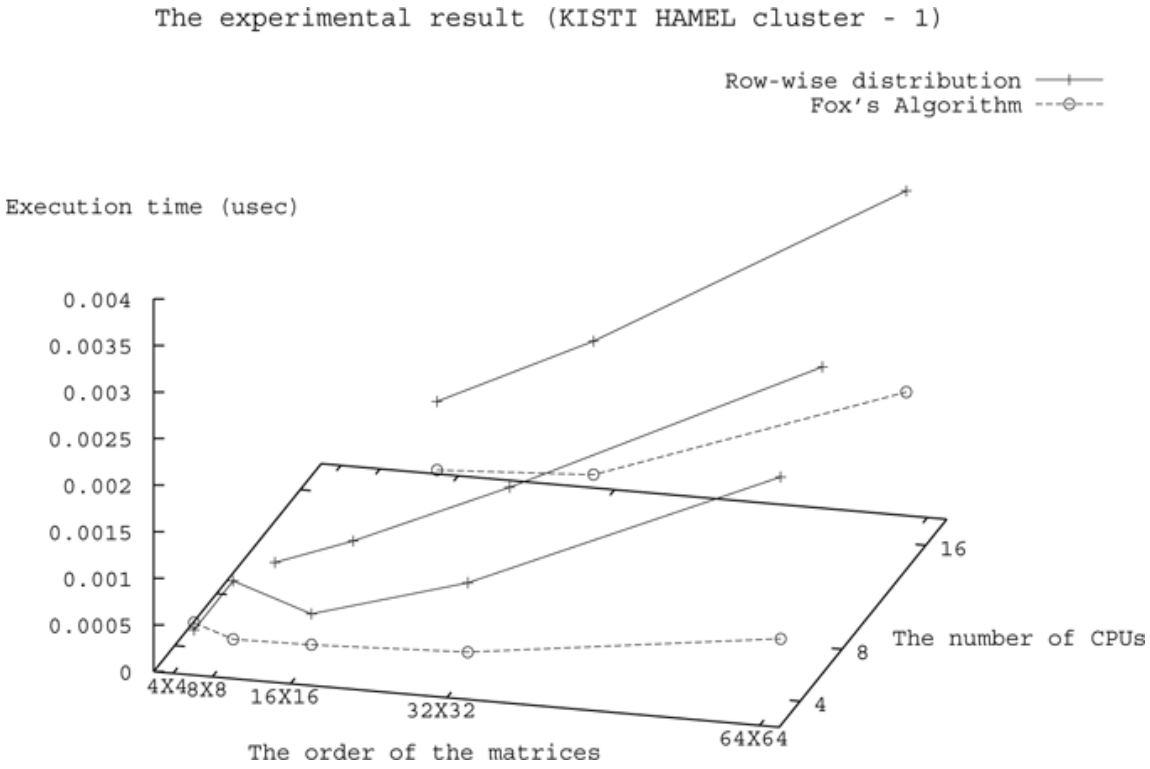

KAIST CAN 클러스터 시스템에서 수행 시간(usec)

| 행렬크기 | 4X4 | 8X8 | 16X16 | 32X32 | 64X64 |
|---|---|---|---|---|---|
| Row-wise | 0.001227 | 0.002002 | 0.004035 | 0.008201 | 0.019313 |
| Fox's | 0.038976 | 0.038593 | 0.038867 | 0.002993 | 0.008947 |

첫번째 프로그램은 행렬크기가 증가함에 따라 수행시간도 증가하는 모습을 보였으나, 두번째 프로그램은 적은 크기에 행렬에서 예상 밖의 성능 저하를 보였다. 32X32 이상의 행렬에서는 두번째 프로그램의 성능이 우수함을 알수 있다.

B. KISTI HAMEL Cluster

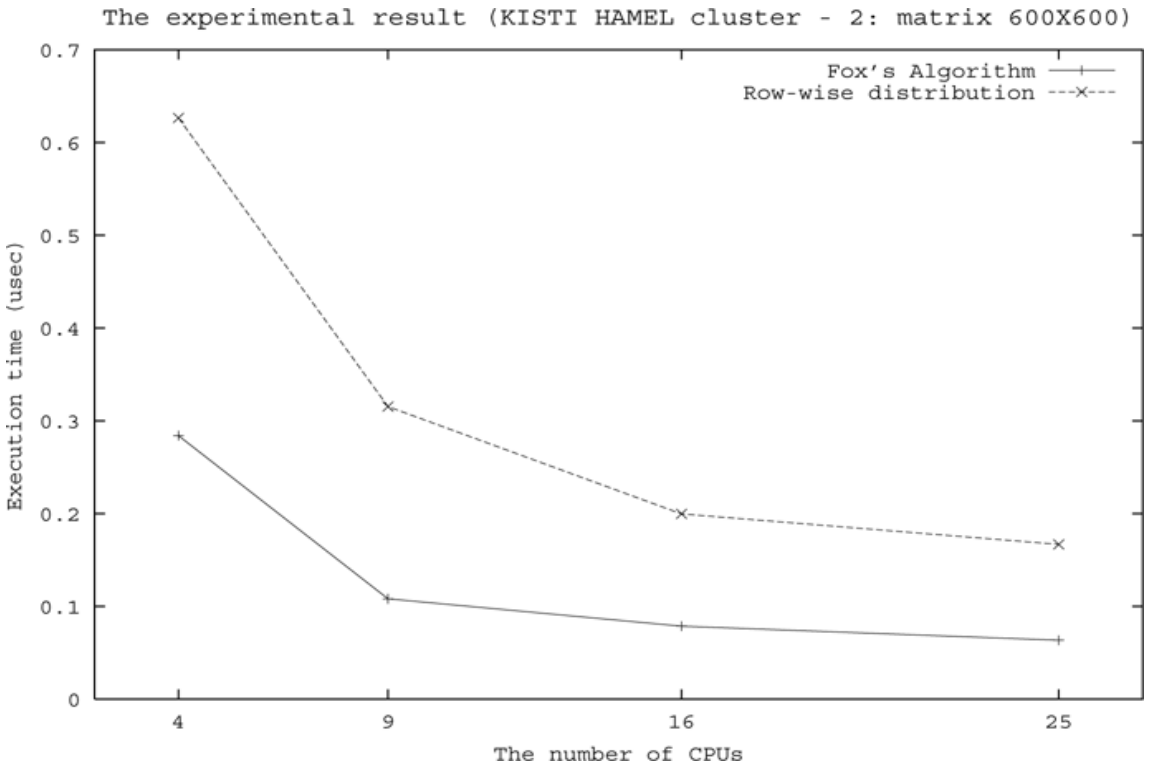
타 클러스터 시스템에서의 성능을 살펴보고자 KISTI의 HAMEL 클러스터 시스템을 사용하였다.(HAMEL 클러스터 시스템은 512개의 Intel Xeon프로세스와 Myrinet으로 구성되어 있슴.)




| Row-wise | | 행렬크기 | | | | |
|---|---|---|---|---|---|---|
| | | 4X4 | 8X8 | 16X16 | 32X32 | 64X64 |
| 노드수 | 4 | 0.000188 | 0.000751 | 0.000471 | 0.000954 | 0.002389 |
| | 8 | | 0.000391 | 0.000697 | 0.001423 | 0.003013 |
| | 16 | | | 0.001078 | 0.001874 | 0.003786 |

| Fox's | | 행렬크기 | | | | |
|---|---|---|---|---|---|---|
| | | 4X4 | 8X8 | 16X16 | 32X32 | 64X64 |
| 노드수 | 4 | 0.000264 | 0.000124 | 0.00014 | 0.000211 | 0.000654 |
| | 16 | | | 0.000342 | 0.000441 | 0.001628 |

앞의 결과는 프로세스 노드를 수를 증가시키면서 수행 시간을 측정한 결과이다. 전반적으로 Fox's Algorithms을 사용한 두번째 프로그램의 성능이 우수함을 확인할 수 있다. 또한 프로세스 노드수가 증가함에 따라 오히려 수행시간이 증가함을 보여주고 있어 병렬 처리를 수행함에 비하여 문제 크기가 적음을 알수 있다. 이에 행렬을 600X600의 크기로 고정하고 프로세스 노드의 수를 변경하면서 수행 시간을 측정하였고 그 결과는 다음과 같다.




| 600X600 행렬에서의 수행시간 (HAMEL 클러스터) | | | | |
|---|---|---|---|---|
| 노드 수 | 4 | 9 | 16 | 25 |
| Row-wise | 0.626529 | 0.315326 | 0.19966 | 0.166849 |
| Fox's | 0.284061 | 0.108163 | 0.078686 | 0.06354 |

## 2.3. 결과 분석

두종류의 시스템을 대상으로 Row-wise distribution 방식의 프로그램과 Checkerboard distribution방식의 Fox's Algorithms을 테스트하였다. 성능면에서는 Fox's Algorithms이 우수한 것을 확인하였으나 다음과 같은 단점 또한 존재한다.

- 프로세스 노드의 수가 항상 $n^2$ 이어야 한다.
- 행렬이 정확하게 정방 행렬로 나누어지지 않는 경우의 처리를 고려해야 한다.(본 보고서에서는 정확하게 나누어지는 경우만 고려 하였슴.)
- 행렬을 배분하고 취합하는 과정이 Row-wise distribution 방식에 비하여 복잡하다.

# 3. Gaussian Elimination using MPI

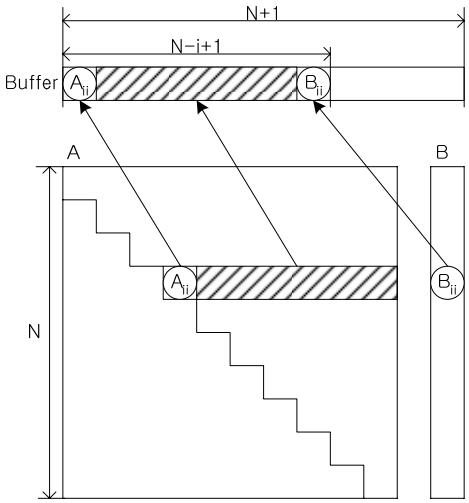
## 3.1. 문제 개요

가우스 소거법 프로그램의 성능 비교를 위하여 단일 프로그램과 Collective Communication(MPI_Bcast)를 사용한 경우와 일대일 통신(MPI_Send & MPI_Recv)를 사용한 두 종류의 병렬 알고리즘을 구현하고 클러스터 시스템에서 테스트하였다.

- 첫번째 병렬 프로그램은 MPI_Bcast를 사용하여 i 단계 작업의 root에 해당하는 프로세스가 디폴트 커뮤니케이터(MPI_COMM_WORLD)에 속한 모든 프로세스에게 데이터($A_i$행)를 전송하도록 하였다. 이때 이미 자신의 데이터에 대한 가우스 소거 작업을 마친 프로세스들은 데이터만 전달받을 뿐, 계산 작업은 수행하지 않는다.

- 두번째 병렬 프로그램은 i 단계 작업의 root에 해당하는 프로세스가 가우스 소거 작업을 끝내지 않은 프로세스들에게 순차적으로 데이터($A_i$행)를 전송하는 방법으로 MPI_Send와 MPI_Recv를 사용하였다.

- 두 병렬 프로그램 모두 $A_{ii}$를 “1” 만든 후, $A_i$행과 $B_i$를 Buffer라고 명명된 1차원 배열에 저장한뒤 MPI의 통신 명령을 이용하여 다른 프로세스들에게 전달하였다. 즉, 아래 그림과 같이 풀고자 하는 행렬의 크기를 N×N이라하면 초기에 N+1의 메모리 영역을 할당하고, i번째 루프에서는 Buffer의 앞에서부터 데이터를 저장한뒤 N-i+1 크기의 데이터를 전송하도록 하였다.




- 이때 root 프로세서의 경우 $A_{ii}$가 “0”인 경우, Buffer의 첫번째 값에 “0”을 입력하여 데이터를 전송하며 데이터를 전달 받은 프로세스들은 Buffer의 첫번째 값이 “0”인 경우, 계산 도중에 $A_{ii}$가 “0”으로 계산된 경우로 간주하여 작업을 중단하게 된다.(즉 이러한 경우의 예로는 행렬값중에 두행의 값이 모두 동일하거나 실수배인 경우를 들수 있다.)

- 파일로 부터 행렬 데이터을 읽은 후, 데이터의 오류 유무(모든 행의 값이 0인 경우, 모든 열의값이 0인 경우)를 확인한 뒤 가우스 소거법을 수행하며, 이 부분은 실행시간 측정에서 제외된다.

- 실행 시간은 행렬이 0번 프로세스 노드로 읽혀진 이후 행렬에 대한 오류 테스트를 수행한뒤부터(즉 0번 프로세스가 데이터를 각 프로세스에 분배하기 전부터), 계산이 완료되어 0번 프로세스 노드가 결과 행렬을 취합하여 화면에 출력하기 전까지를 측정하였다.

## 3.2. 성능 측정

### A. 테스트 환경

FastEthernet을 기본 네트워크로 사용하는 4노드 클러스터 시스템인 4Seasons과 Myrinet으로 연결된 Hamel Cluster에서 성능 측정으로 진행하였다.

| | 4Seasons | Hamel |
|---|---|---|
| CPU | Intel P4 2.4GHz | Intel Xeon 2.8GHz |
| Network | FastEthernet(100Mbps) | Myrinet2000(2Gbps) |
| 실측 Bandwidth* | 89Mbps | 1.7Gbps |
| Compiler | Intel C Compiler(icc) 8.0 | Intel C Compiler(icc) 8.0 |

* 실측 Bandwidth는 mpi환경에서 NetPIPE로 측정한 각 시스템의 통신대역폭이다.

### B. Cluster System I (4Seasons)

실제 노드가 4개인 관계로 노드수가 각각 2와 4인 경우에 대하여 측정하였다.

#### i) 실행 시간

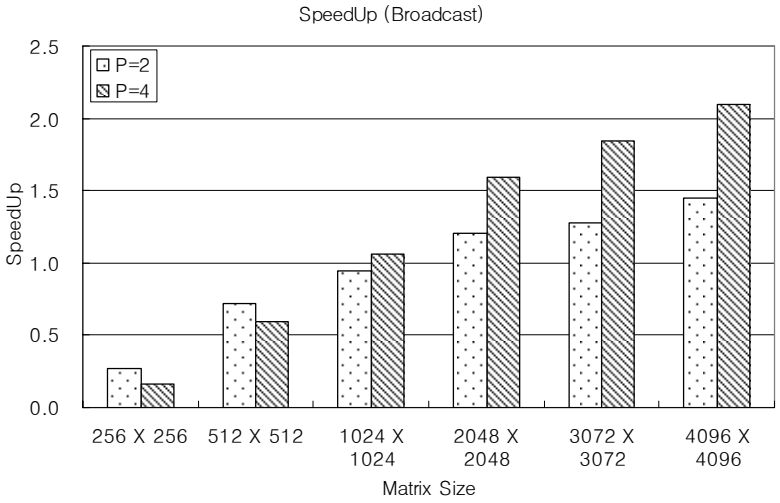
(단위: usec)

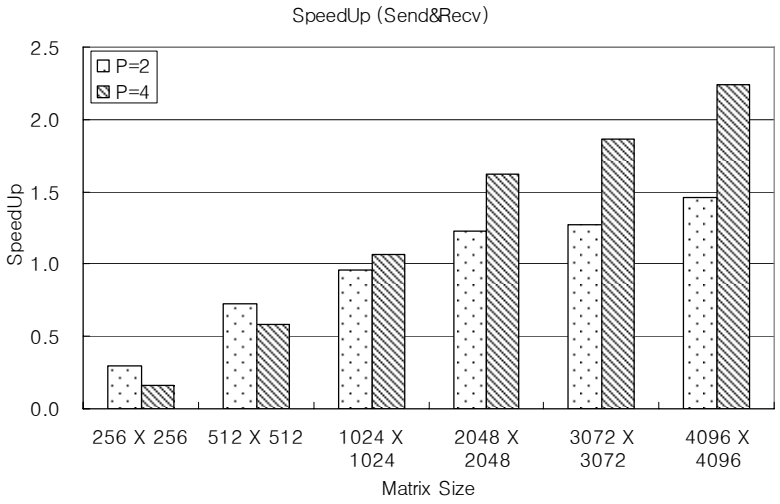
| 4 Seasons Cluster | | | Matrix Size | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 256 X 256 | 512 X 512 | 1024 X 1024 | 2048 X 2048 | 3072 X 3072 | 4096 X 4096 |
| Program type (The number of processes) | Independent | | 10056 | 140607 | 1123586 | 9004752 | 29219553 | 75173640 |
| | Parallel 1 (Broadcast) | 2 | 37502 | 196341 | 1188599 | 7463434 | 22888914 | 51890083 |
| | | 4 | 63336 | 237474 | 1058140 | 5644718 | 15856424 | 35829702 |
| | Parallel 2 (Send&Recv) | 2 | 34436 | 194577 | 1168145 | 7360450 | 22997460 | 51378025 |
| | | 4 | 64029 | 241704 | 1053497 | 5561695 | 15679887 | 33571143 |

#### ii) Speedup (Ts/Tp)

실행 시간을 기준으로 Speedup을 구한 결과는 아래의 테이블과 같다. 두종류의 병렬 프로그램이 미세한 차이는 있는나, 전체적으로 유사한 패턴을 보이고 있으며, 행렬의 크기가 $1024 \times 1024$인 경우부터 병렬 프로그램의 효과를 확인할 수 있다.

| 4 Seasons Cluster | | | Matrix Size | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 256 X 256 | 512 X 512 | 1024 X 1024 | 2048 X 2048 | 3072 X 3072 | 4096 X 4096 |
| Program type (The number of processes) | Parallel 1 (Broadcast) | 2 | 0.268146 | 0.716137 | 0.945303 | 1.206516 | 1.276581 | 1.448709 |
| | | 4 | 0.158772 | 0.592094 | 1.061850 | 1.595253 | 1.842758 | 2.098082 |
| | Parallel 2 (Send&Recv) | 2 | 0.292020 | 0.722629 | 0.961855 | 1.223397 | 1.270556 | 1.463148 |
| | | 4 | 0.157054 | 0.581732 | 1.066530 | 1.619066 | 1.863505 | 2.239234 |



### C. Cluster System II (Hamel)

노드수를 각각 2,,4,8,16으로 변화시키면서 실행 시간을 측정하였다

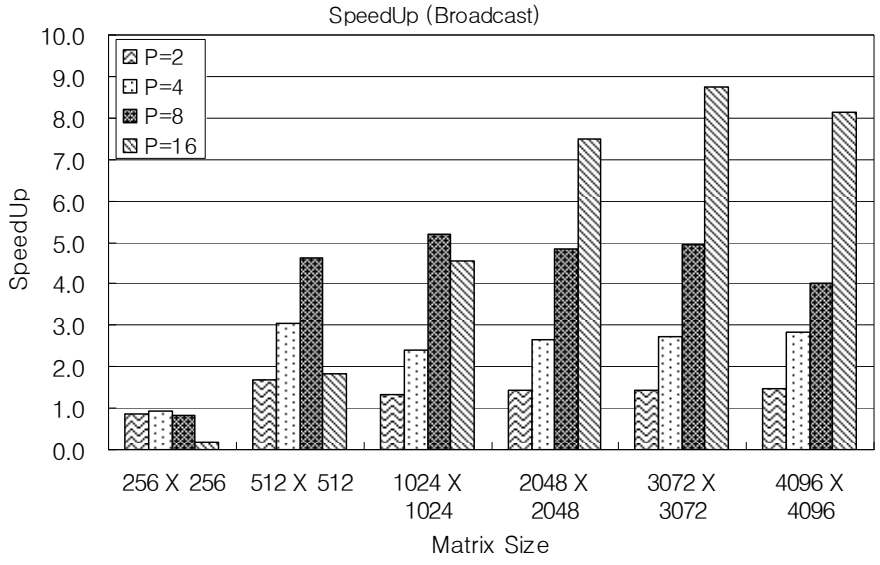
### i) 실행 시간

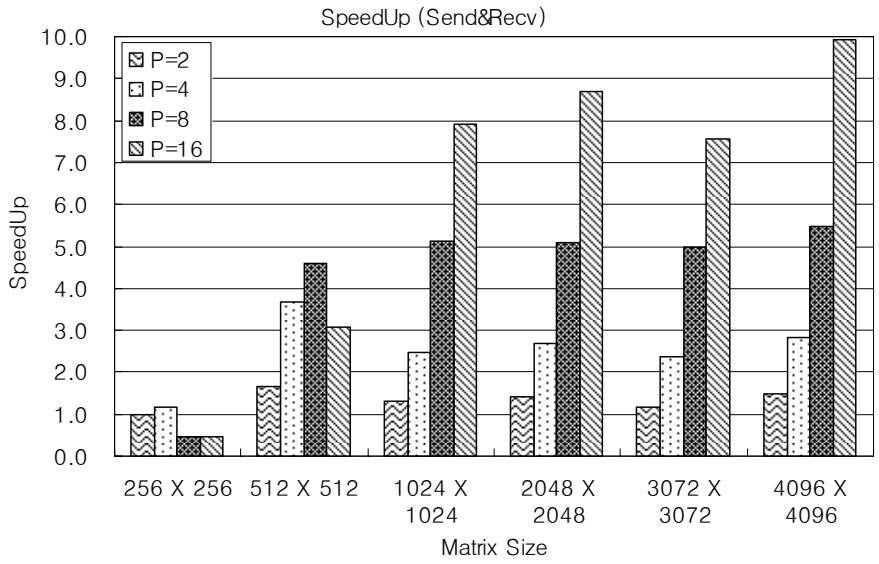
(단위: usec)

| Hamel Cluster | | | Matrix Size | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 256 X 256 | 512 X 512 | 1024 X 1024 | 2048 X 2048 | 3072 X 3072 | 4096 X 4096 |
| Program type (The number of processes) | Independent | | 8416 | 143287 | 1197256 | 9534089 | 32065000 | 77697128 |
| | Parallel 1 (Broadcast) | 2 | 9626.87 | 84486.01 | 903600.93 | 6725386.86 | 22595583.92 | 52609598.88 |
| | | 4 | 8991.96 | 47061.92 | 495053.05 | 3582959.18 | 11790673.02 | 27571587.80 |
| | | 8 | 10150.91 | 30949.83 | 230781.08 | 1970842.12 | 6459363.94 | 19390889.17 |
| | | 16 | 47100.07 | 78767.78 | 264026.88 | 1273880.00 | 3669102.91 | 9536783.93 |
| | Parallel 2 (Send&Recv) | 2 | 8560.18 | 85646.87 | 911479.00 | 6729594.95 | 27211410.05 | 52502141.00 |
| | | 4 | 7277.01 | 38913.97 | 482798.81 | 3542487.14 | 13476819.04 | 27364883.18 |
| | | 8 | 18628.84 | 31178.00 | 233144.04 | 1879456.04 | 6438333.99 | 14208608.87 |
| | | 16 | 18771.89 | 46347.86 | 151240.11 | 1097316.03 | 4231550.93 | 7834068.06 |

### ii) Speedup (Ts/Tp)

| Hamel Cluster | | | Matrix Size | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 256 X 256 | 512 X 512 | 1024 X 1024 | 2048 X 2048 | 3072 X 3072 | 4096 X 4096 |
| Program type (The number of processes) | Parallel 1 (Broadcast) | 2 | 0.874220 | 1.695985 | 1.324983 | 1.417627 | 1.419083 | 1.476862 |
| | | 4 | 0.935948 | 3.044648 | 2.418440 | 2.660954 | 2.719522 | 2.818014 |
| | | 8 | 0.829088 | 4.629654 | 5.187843 | 4.837571 | 4.964111 | 4.006888 |
| | | 16 | 0.178683 | 1.819107 | 4.534599 | 7.484291 | 8.739193 | 8.147100 |
| | Parallel 2 (Send&Recv) | 2 | 0.983157 | 1.672998 | 1.313531 | 1.416740 | 1.178366 | 1.479885 |
| | | 4 | 1.156519 | 3.682149 | 2.479824 | 2.691355 | 2.379271 | 2.839301 |
| | | 8 | 0.451773 | 4.595773 | 5.135263 | 5.072792 | 4.980326 | 5.468314 |
| | | 16 | 0.448330 | 3.091556 | 7.916260 | 8.688553 | 7.577600 | 9.917852 |






4Seasons 시스템의 결과와는 달리 512 × 512 크기의 행렬에서부터 병렬프로그램의 효과를 확인할수 있다. 네트워크의 성능이 개선됨으로서 성능 개선을 볼 수 있는 문제의 크기가 적어짐을 유추할수 있으며, 프로세스의 수를 증가시킴에 따라 성능도 함께 증가함을 확인할 수 있다. 위의 그래프에서는 프로세스의 수가 2,4,8인 경우, 문제의 크기를 증가시키더라도 이미 성능은 어느 정도 포화 상태에 도달한 것으로 보여지며, 프로세스의 수가 16인 경우에는 문제의 크기를 더 크게할 경우 성능이 더 높아질 가능성이 있다는 예측을 할수 있다.

## 3.3. 성능 분석

### A. 클러스터 시스템간의 성능 비교.

서로 다른 클러스터 시스템에서 측정된 실행시간을 바탕으로 비교 분석을 수행하여 보았다. 아래의 테이블은 FastEthernet으로 구성된 4Seasons 클러스터 시스템에서의 실행 시간을 Myrinet으로 구성된 Hamel 클러스터 시스템에서의 실행 시간으로 나눈 값을 보여주고 있다.

| | | 256 X 256 | 512 X 512 | 1024 X 1024 | 2048 X 2048 | 3072 X 3072 | 4096 X 4096 |
|---|---|---|---|---|---|---|---|
| Independent | | 1.194867 | 0.981296 | 0.938468 | 0.944480 | 0.911260 | 0.967521 |
| Parallel 1 | 2 | 3.895557 | 2.323947 | 1.315403 | 1.109740 | 1.012982 | 0.986323 |
| (Broadcast) | 4 | 7.043628 | 5.045990 | 2.137427 | 1.575435 | 1.344828 | 1.299515 |
| Parallel 2 | 2 | 4.022812 | 2.271852 | 1.281593 | 1.093743 | 0.845140 | 0.978589 |
| (Send&Recv) | 4 | 8.798804 | 6.211241 | 2.182062 | 1.569997 | 1.163471 | 1.226797 |

Independent의 경우 노드간의 통신이 없는 단독 프로그램이므로 단일 노드의 성능비를 나타낸다고 볼수 있는데, 문제 크기의 변화와는 무관하게 어느정도 일정한 값을 보여주고 있다. 다만 CPU의 성능으로만 고려할때는 Hamel 클러스터의 CPU 클럭이 빠름에도 불구하고 성능이 약간 저하됨이 특이 사항이다.

병렬 프로그램의 경우 적은 문제의 크기에서는 Myrinet과 FastEternet성능 차이로 인해 두 시스템간의 성능비를 확연히 구분할수 있으나, 문제가 크기가 증가함에 따라서는 비율이 1에 가까운 값으로 바뀜을 볼수 있다. 이는 프로세스의 수가 2, 4로 고정된 상황이므로 문제 크기가 증가하면 Communication to Computation의 값이 작아지면서 두시스템의 성능이 비슷해진다는 점을 보여주고 있다.

### B. 병렬 프로그램간의 성능 비교

집합통신(MPI_Bcast)를 사용한 경우와 일대일 통신(MPI_Send & MPI_Recv)를 사용한 프로그램의 성능을 비교하면 문제의 크기가 작은 경우에는 확실한 판단을 내리기 힘든 결과를 보이고 있다. 그러나 실험에서 문제의 크기가 가장 컷던 4096 × 4096의 경우에는 적게는 0.09%부터 크게는 27%까지 일대일 통신을 쓰는 경우의 성능이 좋음을 확인할수 있다. 즉 본 프로그램의 알고리즘에서는 시간이 지남에 따란 계산 작업에서 제외되는 프로세스가 생겨나므로 이들을 제외하고 통신을 수행하는 방법이 보다 우수한다는 예측을 할수 있는 부분이다.

### C. 결론

두가지 종류의 병렬 프로그램을 두 종류의 클러스터 시스템에서 테스트한 결론을 요약하면 다음과 같다.

- 네트워크의 성능이 좋은 Myrinet으로 구성된 시스템에서 병렬 프로그램의 효과를 볼수있는 문제의 크기가 FastEthernet으로 구성된 시스템에서의 문제

크기보다 상대적으로 작음을 확인할수 있었다.

- 그러나 프로세스의 수를 고정하고 문제의 크기만 증가시키는 경우에는 Communication to Computation의 값이 작아지면서 두시스템의 성능차가 적어짐도 확인하였다.

- 좋은 가독성과 논리의 간단함 그리고 성능등의 이유로 많은 병렬 프로그램에서 집합통신의 사용을 찾아 볼수 있으나, 이번 가우스 소거법에서는 알고리즘의 특성상 일대일 통신을 이용하는 것의 성능이 보다 효율적이었다.

# 4. Gaussian Elimination using OpenMP

## 4.1. 문제 개요

Pipelined Gaussian Elimination을 OpenMP를 이용하여 구현하고 성능을 측정하였다. 결과 분석을 위하여 Row-wise distribution, Block-Block distribution, Loop-level Parallelism 방식으로 프로그램을 작성하였다.

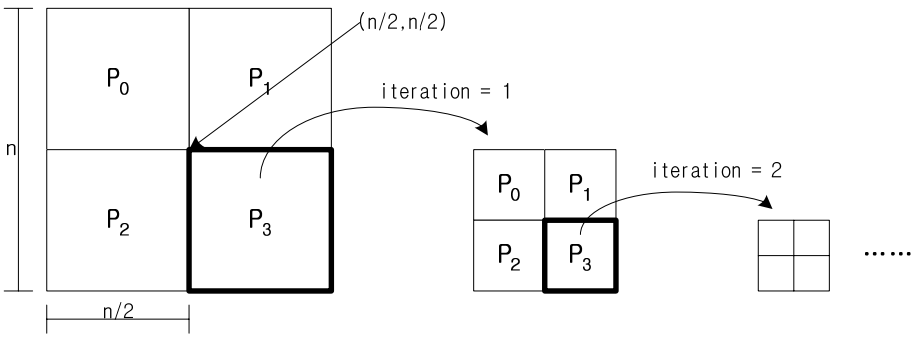
### A. Pipelined Gaussian Elimination: Row-wise distribution 방식

가장 기본적인 방법이라 할 수 있는 Row-wise distribution 방식을 적용하였다.

### B. Pipelined Gaussian Elimination: Block-block(Checkerboard) distribution 방식

파이프라인 기법을 적용한 경우의 성능 측정을 주 목적으로 한다. 기존의 병렬 gaussian elimination에서는 어떤 주어진 시간에 모든 프로세서들이 같은 iteration에 대해 작업을 수행한다. 그러나 pipelined approach에서는 이들 각각의 프로세서들이 asynchronous하게 스케쥴된다. 즉, 각 프로세서들은 필요로하는 데이터가 도착하기를 기다릴 때 외에는 독립적으로 send나 computation 등의 행위를 취할 수 있도록 한다.

2X2의 Threads를 사용하는 것으로 가정하고 Block-Block distribution 방식을 구현하였다. 이 경우 Gaussian Elimination의 특성상 작업의 가장 마지막 단계는 P3(가장 마지막 쓰레드)가 1/4 크기의 하위 행렬에 대한 Gaussian Elimination을 수행하는 것이 된다. Pipelined 기법을 쓰더라도 block-block Distribution에서는 이 마지막 작업을 수행하는 동안 나머지 쓰레드는 계산에 참여하지 않는 상황이 된다. 이에 다음 그림과 같이 변형된 방법을 구현하고 테스트하였다.



즉 반복 분배 횟수를 지정하여 그림과 같이 P3의 하위 행렬에 대한 Gaussian Elimination작업을 다시 모든 쓰레드에게 배분하여 수행하는 방식이다. 즉 Gaussian Elimination의 특성을 고려하여 block-block distribution을 변형시킨 것으로 반복 분배 횟수가 “0” 인 경우에는 순수한 block-block distribution을 수행한 것이 된다.

### C. Loop-level Parallelism 방식

OpenMP의 특징 중, 하나인 Loop을 분할하여 병렬화를 수행하는 기능을 이용하였다.

## 4.2. 성능 측정

### A. 테스트 환경

테스트에 사용된 시스템은 Intel Xeon 2.8GHz를 사용하는 SMP 서버로서 Hyper-Threading 기능을 사용하였고, Intel C Compiler(icc) 8.0를 이용하였다.

### i) 실행 시간

(단위: usec)

| | Matrix Size | | | | |
|---|---|---|---|---|---|
| | 512X512 | 1024X1024 | 2048X2048 | 3072X3072 | 4096X4096 |
| Single(Independent) | 145912 | 1162707 | 9138802 | 30065493 | 76046749 |
| Row-wise(Thrs=5) | 3764557 | 7818094 | 17945323 | 37900040 | 71280472 |
| Row-wise(Thrs=4) | 83843 | 1059113 | 8344565 | 27765954 | 65726548 |
| Loop-level Parallel(Thrs=4) | 102546 | 1152395 | 8400522 | 27705305 | 65322866 |
| Block-Block(Thrs=4,Itr.=0) | 132826 | 1236698 | 8986309 | 30330146 | 71023097 |
| Block-Block(Thrs=4,Itr.=1) | 103152 | 1228923 | 8891498 | 30016511 | 69028609 |

### ii) Speedup (Ts/Tp)

| | Matrix Size | | | | |
|---|---|---|---|---|---|
| | 512X512 | 1024X1024 | 2048X2048 | 3072X3072 | 4096X4096 |
| Row-wise(Thrs=5) | 0.038759408 | 0.148720008 | 0.50925815 | 0.793283938 | 1.066866519 |
| Row-wise(Thrs=4) | 1.740300323 | 1.097812037 | 1.095180156 | 1.082818656 | 1.157017238 |
| Loop-level Parallel(Thrs=4) | 1.422893141 | 1.008948321 | 1.087885015 | 1.085189028 | 1.164167368 |
| Block-Block(Thrs=4,Itr.=0) | 1.098519868 | 0.940170519 | 1.016969481 | 0.991274259 | 1.070732652 |
| Block-Block(Thrs=4,Itr.=1) | 1.414533892 | 0.946118675 | 1.027813536 | 1.001631835 | 1.101670019 |

### 4.3. 성능 분석

Threads수가 4인 경우의 Row-wise distribution방식이 가장 좋은 성능을 보였으며 Loop-level Parallelism과 유사한 성능을 보였다. Block-Block의 경우, 원래 방식(반복 없슴)보다 수정된 방식(1회 반복)에서 약간의 성능 개선을 확인할 수 있었으나, Row-wise distribution방식보다 좋은 성능을 보이지는 않았다.

# 5. 결 론

클러스터 시스템 환경에서 MPI와 OpenMP를 사용한 응용 프로그램 레벨 테스트를 진행하였다. 응용 프로그램의 병렬화 방식에 맞는 병렬 라이브러리의 선택도 중요하지만, 시스템의 구성 조건 또한 병렬 라이브러리의 선택에 영향을 미칠수 있다는 점을 확인하였다.

결국 시스템 구성, 시스템 특징, 문제 크기 및 응용프로그램의 특성이 전체 성능에 영향을 미치므로 이를 복합적으로 고려하여 성능 테스트 및 최적화를 진행하여야 할 것이다.

# 참 고 문 헌

[1] Peter S. Pacheco,"Parallel Programming with MPI," Morgan Kaufmann Publishers, 1997

[2] The Message Passing Interface (MPI) standard, http://www-unix.mcs.anl.gov/mpi/

[3] OpenMP Official Web Site, http://www.openmp.org/blog/

[4] Rajkumar Buyya, "High Performance Cluster Computing: Architectures and Systems," Prentice Hall PTR, 1999

[5] Gregory Pfister, "In Search of Clusters," Prentice Hall PTR, 1997