클러스터 시스템 성능 측정

김호중 (Hojoong Kim)
한국과학기술원 전자전산학과 전산학전공

2003년 7월 28일

1. HPL 벤치마크

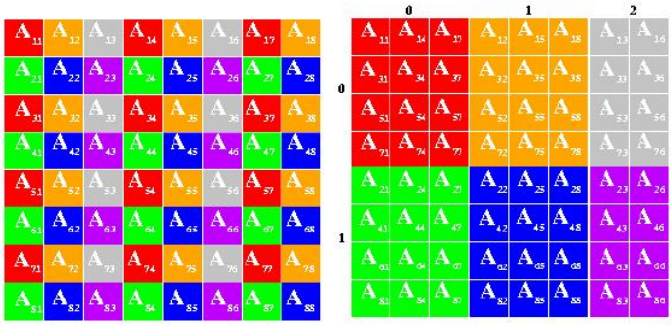
1.1 소개

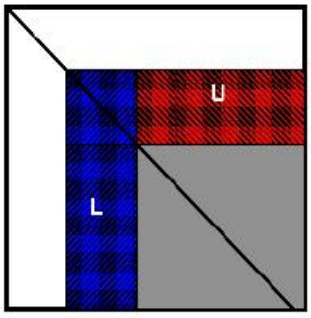
HPL 벤치마크[1]는 LINPACK[2] 벤치마크를 병렬화 하여 분산 시스템에서 수행 가능하도록 구현한 것이다. LINPACK 벤치마크는 슈퍼컴퓨팅 top500 사이트[3]의 기본 성능 측정 수단으로 활용되는 등 컴퓨터 구조 분야에서 널리 사용되는 벤치마크 프로그램이다. 그러나 원래 LINPACK 벤치마크는 여러 개의 프로세서가 하나의 커다란 메모리 공간에 접근하여 계산을 수행하도록 구성 되어 있다. 따라서 프로세서와 메모리 들이 물리적으로 떨어져 있는 클러스터 시스템에서는 LINPACK 벤치마크를 수행하기 곤란하다. HPL 벤치마크는 MPI를 사용하여 LINPACK 프로그램을 병렬화 함으로써 이러한 제약 조건을 제거하여 준다. HPL 벤치마크의 측정 결과는 슈퍼컴퓨팅 top500 사이트 및 클러스터 컴퓨팅 top500 사이트[4]에서 LINPACK 결과와 동일한 값으로 인정된다.

1.2 알고리즘 설명

이 소프트웨어 패키지는 n차 선형방정식 Ax=B 의 n by n+1 계수행렬(coefficient matrix)을 LU 인수분해(LU factorization)로 계산하여 푼다. 따라서, 방정식 Ax=b가 LUx=b로 되고 새로운 n by 1 행렬 y를 정의할 때 Ux = y 가 된다. 이를 사용하여 Ax=b를 Ly=b로 고쳐 Ly=b에서 y를 계산하고 이를 Ux=y에 대입하여 x의 해를 구한다.

데이터는 부하 분산을 위해서 2차원 블록 사이클 방법(block-cycle scheme)을 사용하여 P by Q의 프로세스들 그리드로 분배된다. n by n+1 계수행렬은 먼저 논리적으로 NB by NB 블록으로 분할된다.

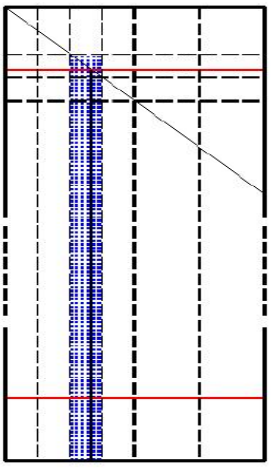




right-looking variant 방법이 LU 인수분해의 메인 루프를 위해 선택되면, 이는 루프의 각 반복마다 NB 컬럼 만큼씩 인수분해 되는 것을 의미한다. 즉, 이 계산은 데이터 분배를 위해 사용되는 블록 크기 NB 단위로 논리적으로 분할된다.

1.3 Panel Factorization

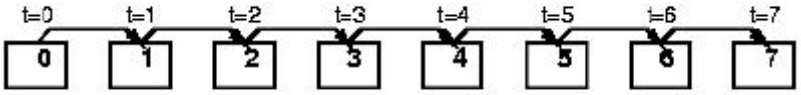
메인 루프의 각 반복에 있어 분포 방법의 카르테시안 속성으로 인하여 각 패널 인수분해가 프로세스의 한 컬럼에서 일어난다. 계산의 특정 부분은 전체 알고리즘의 임계 경로(critical path)가 되어 성능에 큰 영향을 미칠 수 있다. 사용자는 재귀적 변형에 기반한 세가지 행렬 곱셈(Crout, left- and right-looking)방법을 선택할 수 있다. 이 벤치마크 소프트웨어는 또한 사용자로 하여금 현재 패널을 얼마나 많은 하부 패널로 분할하여야 하는지에 관하여도 선택할 수 있도록 한다. 더 나아가서 실행중(run time)에 루프의 반복을 중단하는 기준이되는 임계값도 설정할 수 있도록 한다.



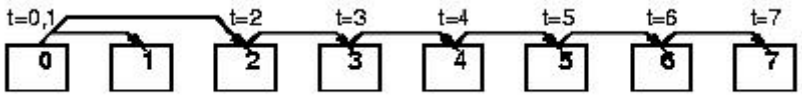
1.4 Panel Broadcast

일단 패널의 인수분해가 계산되면, 컬럼의 패널이 다른 프로세스 컬럼들에게 broadcast된다. 가능한 broadcast 알고리즘으로는 여러 가지가 있으며, HPL 벤치마크에서는 6가지 방법 중 한 가지를 선택할 수 있다. 여기서는 편의를 위해 broadcast를 수행하는 root 프로세스를 프로세스 0이라 가정하고 → 기호는 메시지를 전달하는 것을 의미한다.

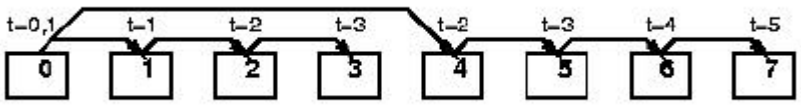
- Increasing-ring : 0→1; 1→2; 2→3과 같은 방식으로 계속된다. 고전적인 알고리즘으로 한 프로세스는 메시지 하나만을 전달하지만, 전체 broadcast 수행 시간이 길어진다.



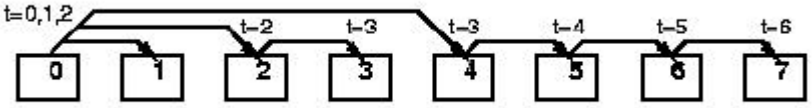
- Increasing-ring(modified) : 0→1; 0→2; 2→3 과 같은 방식이다. 프로세스 0은 두개의 메시지를 전송하고 프로세스 1은 메시지를 받기만 한다. 이 알고리즘은 대체로 우수한 성능을 보인다..



- Increasing-2-ring : Q개의 프로세스들은 0→1과 0→Q/2로 전달하는 두 부분으로 나뉘어진다. 프로세스 1과 Q/2는 1→2, Q/2→Q/2+1; 2→3, Q/2+1→Q/2+2와 같이 두 ring의 root로 행동한다. 따라서 프로세스 0이 2개의 메시지를 보내는 비용으로 마지막 프로세스가 패널을 받을 수 있으므로 시간을 줄일 수 있다.



- Increasing-2-ring (modified) : Increasing-2-ring과 비슷하게 먼저 0→1로 전달하고 남은 Q-1 프로세스들은 0→2와 0→Q/2의 두개의 같은 부분으로 나뉘어진다. 그러면, 프로세스 2와 Q/2는 2→3, Q/2→Q/2+1; 3→4, Q/2+1→Q/2+2 와 같은 방법으로 두 ring의 root로 행동한다. 이 알고리즘은 increasing ring modified 변형과 더불어 가장 우수한 알고리즘 중 하나이다.



- Long (bandwidth reducing) : 이 알고리즘은 이전의 알고리즘들과는 반대되는 것으로 연산에 관련된 모든 프로세스들을 동기화한다. 메시지는 Q개의 같은 크기의 조각들로 쪼개어지고 Q개의 프로세스들에게 분산된다.
  분산 절차는 이진 트리 형식과 rolling phase를 사용한다. 조각들은 Q-1스텝부터 시작한다.홀수 스텝에서 0↔1, 2↔3, 4↔5와 같이, 그리고 짝수 스텝에서 Q-1↔0, 1↔2, 3↔4, 5↔6와 같이 메시지 전송이 이루어진다.
  많은 메시지들이 교환됨에도 불구하고 통신의 전체 크기는 Q에 독립적인데 이 점이 이 알고리즘을 특별히 큰 메시지에도 적합하도록 만드는 것이다. 이 알고리즘은 노드들이 매우 빠른 경우 그리고 네트워크가 상대적으로 느린 경우 적합하다.
- Long (bandwidth reducing modified) : 처음에 0→1로 먼저 전송되는 것만 제외하고 위의 알고리즘과 동일하다.

1.5. Look-ahead

패널이 broadcast되거나 혹은 그 도중에 그 동안 사용한 하위 행렬들이 look-ahead 파이프에서 마지막 패널을 사용하여 갱신된다. 패널 인수분해는 임계 경로상에 속해 있다. k번

째 패널이 인수분해 되고 broadcast될 때 다음 번 가장 급한 작업이 인수분해 되고 k+1번째 패널로 broadcast된다. 이 방법은 종종 look-ahead 또는 send-ahead라 불린다.

HPL 벤치마크에서는 다양한 look-ahead depth를 설정할 수 있도록 한다. 관례적으로 depth 0은 look ahead를 사용하지 않는다. 이런 경우 사용된 하위 행렬은 현재 broadcast하는 패널에 의해서 갱신된다. 일반적으로 look ahead depth=1인 경우 가장 좋은 성능을 낸다. Look-ahead는 추가적으로 메모리를 소비한다.

1.6 Update

Look-ahead pipe에서 마지막 패널에서 사용된 하위 행렬의 갱신은 두 단계로 나뉘어진다. 첫번째로 pivot이 현재 행 패널 U(Upper triangular)에 적용되어야 한다. U는 열 패널의 상삼각행렬에서 풀어져야 한다.

1.7 Backward substitution

인수분해가 모두 끝나면 이제 선형방정식을 풀기 위해 후위 치환(backward substitution)이 수행 되어야 한다. 이를 위해서 우리는 depth의 look-ahead를 선택한다. 한번에 Q*nb씩 풀기 위해서 프로세스 행에서 오른쪽 부분이 decreasing-ring 형태로 전달된다. 각 단계에서 계수행렬의 오른쪽 부분이 갱신된다. 위의 프로세스는 행렬 A의 현재 대각선 블록을 소유하는 것으로 먼저 x의 마지막 NB 조각을 갱신하고 이를 이전의 프로세스 컬럼에 전달한다. 그런 다음 decreasing-ring 형태로 이전의 프로세스 컬럼으로 전송한다. 선형방성식의 해는 모든 프로세스 행에 복사되어 남는다.

1.8 Checking the solution

얻어진 결과가 정확한지를 확인하기 위해서 입력 행렬과 오른쪽 부분이 다시 생성된다. 세가지 나머지가 계산되고 이들의 크기가 임계값보다 작을 때 수치적으로 정확하다고 판단한다. 아래 수식에서 eps는 상대적인 기계 정확도(relative machine precision)이다.

- ||Ax-b||_oo / (eps * ||A||_1 * N)
- ||Ax-b||_oo / (eps * ||A||_1 * ||x||_1)
- ||Ax-b||_oo / (eps * ||A||_oo * ||x||_oo)

2. 성능 평가 환경

2.1 시스템 구성

본 연구에서 성능을 측정한 클러스터 시스템은 <표 1>과 같이 IBM x440 4-way SMP 노드 4대로 구성되어 있다.

IBM x440 (4노드)	프로세서	Intel Xeon 1.5GHz 4-way SMP 256KB L2 Cache Memory 1MB L3 Cache Memory 32MB L4 Server Accerlator Cache Memory
	메모리	2048MB SDRAM
	I/O	PCI-x 33/66/133MHz
	NIC	3C996B-TX Gigabit Ethernet NIC
네트워크	Switch	3C17701 24-ports Gigabit Ethernet Switch
S/W	운영체제	Linux 2.4.20 (Suse Linux 8.2)
	MPI	MPICH 1.2.5 arch=ch_p4, dev=shmem
	수학 package	Automatically Tuned Linear Algebra Software (ATLAS) 3.4.1 prebuilt library, Intel P4 using SSE2

<표 1> 시스템 구성

SMP를 효율적으로 활용할 수 있는 커널 버전인 Linux 2.4 kernel을 설치하였으며 공유메모리와 TCP/IP를 동시에 사용하여 메시지를 전달할 수 있는 MPI 구현인 MPICH 1.2.5 버전을 사용하였다.

HPL benchmark와 같은 공학 계산 프로그램은 MPI를 기반으로 동작하며 정밀한 수학 계산을 위한 library package를 필요로 한다. 이러한 수학 library package에는 BLAS[5]나 VSIPL[6] 등이 있으며 본 연구에서는 BLAS library를 자동으로 생성하는 프로그램인 ATLAS[7]를 사용하였다.

2.2 벤치마크 환경 설정

HPL 벤치마크에서는 앞서 알고리즘에서 언급한 바와 같이 다양한 옵션을 줄 수 있다. 이러한 옵션들 중 성능 측정 결과에 큰 영향을 미치는 것은 문제 크기 N을 비롯하여 P, Q, NB 등의 값이다. 노드 수와 프로세스 수를 증가시켜 가며 다양한 실험을 수행한 결과 대부분의 경우에서 NB=128인 경우 가장 높은 성능 측정 결과를 얻었다. 또 전체 n개의 프로세서를 P×Q로 나누는 경우 정방형에 가깝게(P=Q) 나눌 때 또는 P가 Q보다 약간 작은 경우 가장 좋은 결과를 얻었다.

벤치마크 수행 시 사용한 옵션들을 <표 2>에 정리하였다.

옵션	값	비고
N	다양하게 변화시키며 측정	문제 크기
P, Q	프로세스 수에 종속됨	정방형에 가깝게 할당
NB	128	64~256 사이에서 최적
PFACT	Crout	
NBMIN	4	
NDIV	2	
RFACT	Crout	
Broadcast	Increasing ring (modified)	
Lookahead depth	0	
SWAP	Mix	Threshold = 64
L	Transposed form	
U	Transposed form	
Equilibration	yes	
Memoy alignment	8 double precision word	64바이트 단위 정렬

<표 2> HPL benchmark에서 사용하는 옵션들

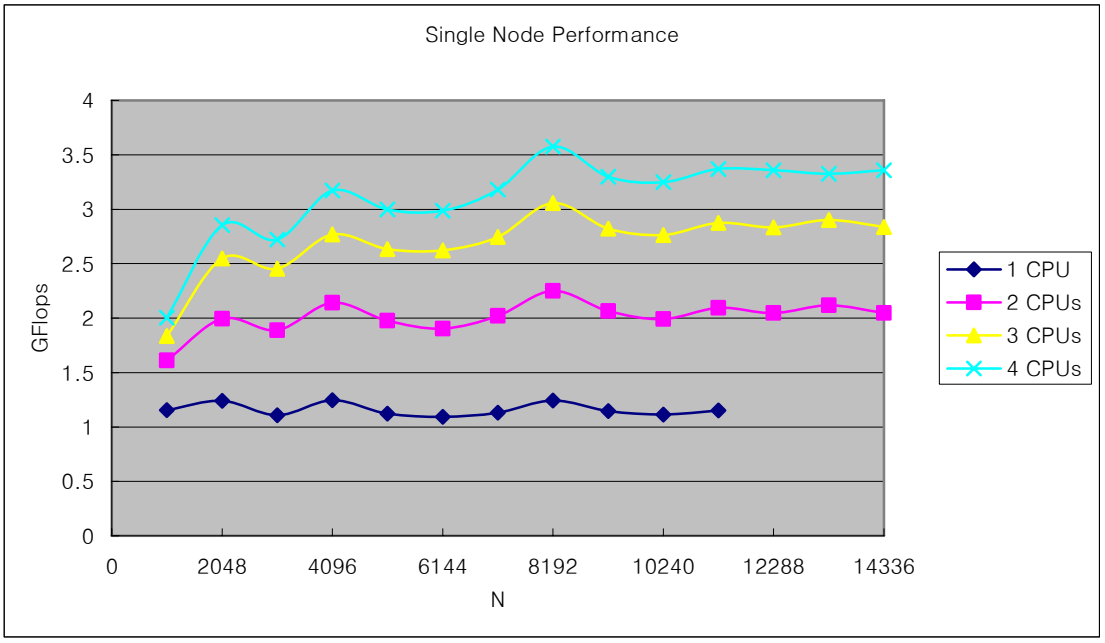
3. 성능 평가

2.1 단일 노드 성능 측정 결과

4-way SMP 단일 노드의 성능을 측정한 결과는 아래와 같다.

프로세서 개수	1 CPU	2 CPUs	3 CPUs	4 CPUs
P	1	1	1	1
Q	1	2	3	4

<표 3> 단일 노드 성능 측정 환경

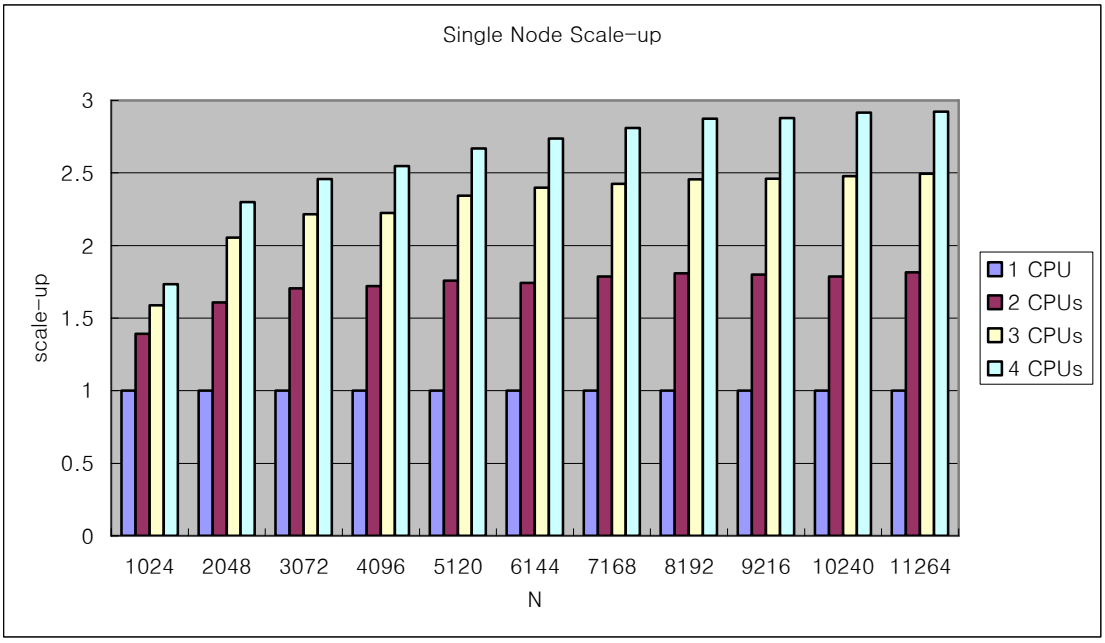


<그림 1> 4-way SMP 단일 노드 성능 측정 결과

한 개의 CPU를 사용하여 얻을 수 있는 성능은 N=4096, 8192일 때 최대 1.24GFLOPS였다. 각각의 CPU는 1.5GHz이므로 이러한 성능 측정 결과는 이론적인 성능에는 못미치는 수치이다. 이는 OS 및 벤치마크 소프트웨어의 최적화가 덜 되었기 때문이라고 할 수 있다.

한편 단일 CPU의 성능은 문제 크기 N을 증가시켜도 크게 좋아지지 않는 것을 볼 수 있다. 여러 개의 프로세스가 각기 다른 CPU에 할당되어 동작하는 경우 문제 크기 N이 작다면 실제 계산하는 시간에 비하여 프로세스간 통신 시간이 길어져 CPU의 성능을 충분히 활용하지 못하지만 하나의 CPU에서 모든 계산을 수행하는 경우 이러한 통신 오버헤드가 없으므로 CPU의 성능을 충분히 활용할 수 있다. 그러나 실제로는 Linux 내의 다른 프로세스들과의 context switching 등으로 수행 중인 프로세스가 다른 CPU로 옮겨서 다시 수행되는 현상이 관찰되었고 이로 인하여 이론적인 수치에 못미치는 결과를 얻은 것으로 보인다.

SMP를 사용함으로써 얻는 성능 향상을 비율로 나타내면 아래 <그림 2>와 같다.



<

그림 2> 단일 SMP 노드에서의 성능 향상 비율

<그림 1>과 <그림 2>에서 보듯이 SMP를 활용함으로써 시스템의 최대 성능이 증가하며 특히 문제 크기 N이 증가할수록 보다 높은 성능을 얻는다는 것을 알 수 있다. 그러나 그 성능 증가 폭은 CPU 수를 추가할수록 줄어든다. 2개의 CPU를 사용하는 경우 하나의 CPU를 사용하는 경우에 비하여 N>2048일 때 약 1.7~1.8배, 평균적으로 1.77배 정도로 비교적 성능 향상 폭이 높다. 그러나 3개의 CPU를 사용하는 경우 N>2048일 때 약 2.2~2.5배, 평균적으로 2.39배의 성능 향상을 보였으며 4개의 CPU를 사용하는 경우 N>2048일 때 약 2.3~2.9배, 평균 2.76배의 성능 향상을 보였다. 이처럼 CPU의 개수가 늘어날수록 성능 향상이 줄어드는 이유는 여러 프로세스 간의 통신을 위한 동기화 시간이 늘어나며, 여러 개의 CPU가 동시에 메모리에 접근함으로써 발생하는 bus contention 등 오버헤드가 증가하기 때문이다.

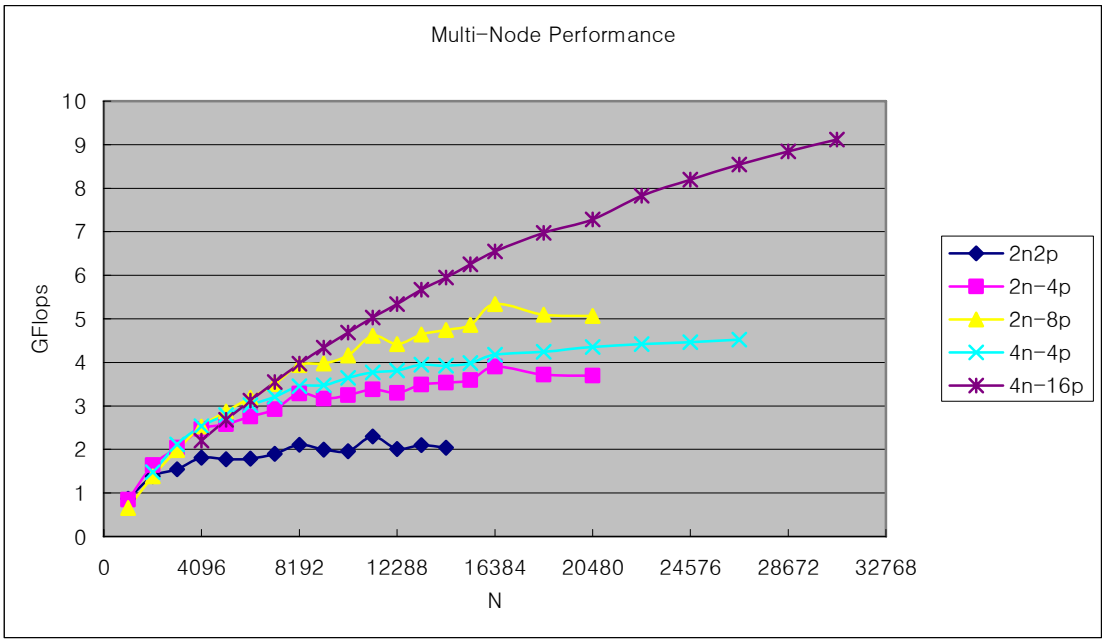
한편 <그림 1>에서 문제 크기 N=2048, 4096, 8192일 때 대체로 벤치마크 성능이 높게 나타났다. $N=2^k$ (k=정수)일 때는 전체 행렬을 각 프로세스에 골고루 배분할 수 있으므로 프로그램 수행의 효율성이 높아지는 것이다. 이를 통하여 시스템의 특성을 고려하여 응용 프로그램을 작성하는 것이 시스템의 성능에 큰 영향을 미친다는 것을 확인할 수 있다.

2.2 다중 노드 성능 측정 결과

IBM x440 4-way SMP 노드를 Gigabit Ethernet으로 연결하여 클러스터를 구성하고 성능을 측정하였다. 한 CPU 당 최대 한 개의 프로세스를 수행하였으며 같은 노드 상의 프로세스끼리는 공유메모리를 사용하여, 다른 노드 상의 프로세스와는 TCP/IP를 사용하여 통신한다.

노드 수	2			4	
프로세서 수	2	4	8	4	16
P	1	2	2	2	4
Q	2	2	4	2	4

<표 4> 다중 노드 성능 측정 환경



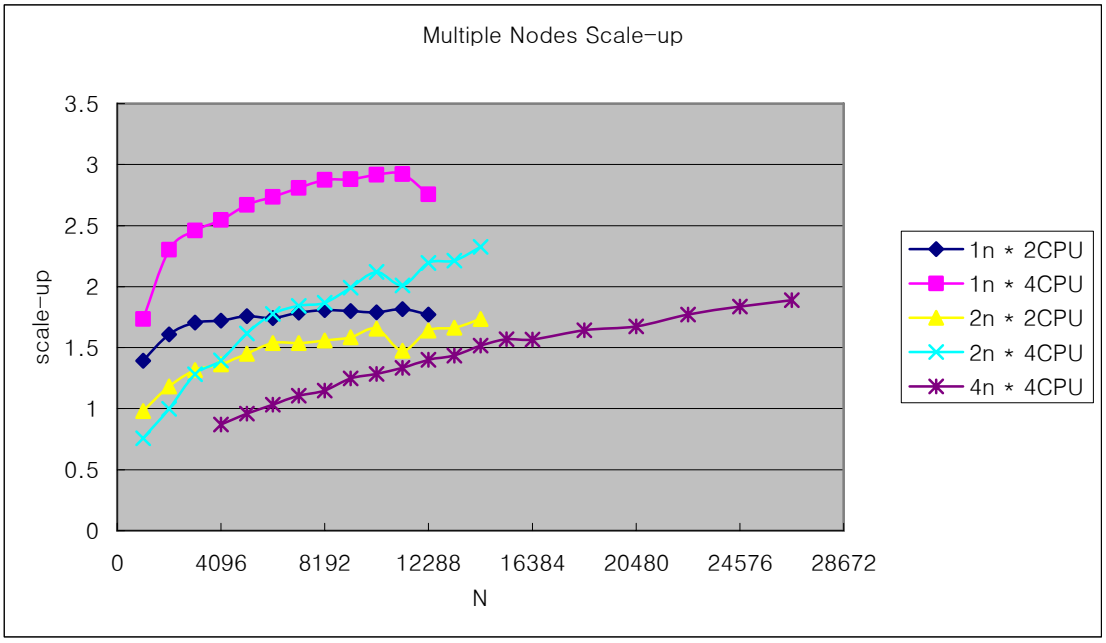
<그림 3> 다중 노드 성능 측정 결과

문제 크기 N은 30720까지 증가시켜 가며 측정하였다. 4노드 16프로세서의 경우 N이 증가함에 따라 성능이 좀 더 증가할 수 있을 것으로 생각된다. 그러나 문제 크기 N = 32768인 경우 전체 행렬의 크기는 N * N * (size of double) = 32K * 32K * 8 = 8GBytes가 된다. 본 시스템은 노드 당 2GBytes, 총 8GBytes의 메모리를 가지고 있으므로 행렬의 크기를 8GBytes로 잡는 경우 OS가 차지하는 기본 메모리와 기타 프로그램 수행에 필요한 메모리 등을 포함하면 전체 사용 가능한 메모리보다 커지게 되고, 따라서 thrashing이 발생하여 성능 측정 결과는 심각하게 저하될 것이다.

4개의 노드를 사용하는 경우 한 노드에 4개의 프로세스를 수행하는 경우 하나의 프로세스를 수행하는 경우보다 약 2배정도의 성능 향상을 가져왔다. 이는 단일 노드 시스템에서 평균 2.76배의 성능 향상을 보인 것에 비해 적은 수치이다. 네트워크 contention 및 프로세스 간의 동기화 오버헤드가 더 커지기 때문이다.

4노드 4프로세스의 경우 최대 4.5GFlops, 16프로세스의 경우 최대 9.12GFlops의 성능을 얻을 수 있었다.

한편 2개의 노드에서 노드당 각각 2개씩 4개의 프로세스를 수행한 경우(2n-4p)보다 4개의 노드에서 노드당 각각 하나씩 4개의 프로세스를 수행한 경우(4n-4p)의 성능이 더 높다. 따라서 SMP 내에서의 통신 오버헤드가 Gigabit Ethernet의 통신 오버헤드보다 높다고 할 수 있다. 따라서 Linux 상에서 SMP의 효율적인 지원 방법이 필요하다.



<그림 4> SMP의 성능 향상 비율

<그림 4>는 SMP를 사용할 때의 성능 향상 비율을 나타낸 것이다. 각각 한 노드당 한 프로세스만을 실행시킬 때와 비교하였다. 즉 4n * 4CPU의 성능 향상은 4n * 1CPU와 비교하였고 1n * 2CPU의 성능 향상은 1n * 1CPU와 비교하였다.

노드 수가 증가함에 따라 노드 당 프로세스의 수를 증가시킴으로서 얻을 수 있는 성능 향상의 비율이 하락한다. 따라서 본 연구 환경과 같은 SMP 클러스터에서는 노드 수를 증가하여 얻을 수 있는 성능 향상에는 한계가 있다는 것을 알 수 있다. 이러한 현상이 발생하는 이유는 기존 클러스터 환경이 SMP를 지원하는데 적합하지 않기 때문이며 따라서 OS를 비롯하여 소프트웨어적으로 최적화하고 SMP를 효율적으로 지원하는 방법을 찾는 일이 중요하다.

4. 결론

본 클러스터 시스템에서 HPL 벤치마크를 수행한 결과 IBM x440 4-way SMP 4노드 16프로세서 시스템에서 9.12GFlops의 성능을 얻을 수 있었다. 이 값을 Pentium-III 850MHz 96프로세서 시스템의 프로세서 속도에 맞추어 산술적으로 환산하면 약 31GFlops이다. 예상한 성능인 28.75GFLOPS와는 약간의 오차가 있다. 이러한 오차가 발생하는 이유는 프로세서의 성능 차이 외에도 네트워크 및 소프트웨어의 성능 차이가 생기기 때문이다.

시스템 구성	실제 구축한 클러스터 시스템	비교 대상 클러스터 시스템
프로세서	Intel Xeon 1.5GHz 4-way SMP	Pentium III 850MHz dual CPU
메모리	2048MB	512MB
네트워크 카드	Gigabit Ethernet	Myrinet LANai 9.2

IBM x440은 우수한 하드웨어를 사용하는 만큼 높은 성능을 얻을 수 있을 것으로 기대하였으나 예상보다 낮은 성능을 나타내었다. 그 이유는 SMP 시스템의 특성에서 찾아야 할 것이다. SMP의 특성상 각 CPU에 할당된 프로세스들이 같은 메모리 버스와 I/O 버스를 통해 자원에 접근하므로 contention이 발생한다. 또한 프로세스 간의 통신을 위한 동기화 오버헤드가 커진다. CPU의 개수를 늘릴수록 성능 향상의 폭이 줄어드는 결과로부터 이러한 사실을 확인할 수 있다.

SMP의 성능을 최대한 활용하기 위해서는 소프트웨어의 최적화가 필요하다. 본 연구에서 사용한 Linux kernel 2.4.20은 SMP를 지원하지만 최적화 되지는 않은 OS이다. 또한 HPL 벤치마크에서 전체 행렬을 각 프로세스에 나누어 주는 방식에 따라 물리적으로 같은 노드에 위치한 프로세스 간의 통신 성능을 향상시킬 수 있을 것이다.

한편, 프로세서 수의 증가에 따른 성능 증가 그래프를 통하여 시스템의 성능이 거의 선형적으로 증가함을 확인하였다. 또한 2-way SMP를 사용함으로써 16노드의 경우 단일 프로세서에 비해 1.4배 가량의 성능 증가를 얻을 수 있었다. 2-way SMP의 도입 가격을 고려할 때, 가격 대비 성능에서 경쟁력이 있다고 평가된다.

4. 참고 문헌

[1] http://www.netlib.org/benchmark/hpl
[2] http://www.netlib.org/linpack/
[3] http://www.top500.org
[4] http://cluster.top500.org
[5] http://www.netlib.org/blas
[6] http://www.vsipl.org
[7] http://www.netlib.org/atlas