제4회 리눅스 공동체 세미나 강의록

D-3세션
리눅스 클러스터링

리눅스를 이용한 클러스터 아키텍쳐 구성

1. 리눅스 클러스터 아키텍쳐에 관하여
2. 리눅스 클러스터의 실제적인 구성
3. MPI를 사용한 리눅스 클러스터 프로그래밍

● 김세용
skim@kunja.sejong.ac.kr

● 게시판 주소
http://seminar.klug.or.kr/inform/teacher/index.php3?ses=d_3

1 리눅스 클러스터 아키텍처에 관하여

Premise : 리눅스 클러스터는 기본적으로 클러스터 아키텍처이고, 클러스터는 병렬 컴퓨터 아키텍처의 일종이다. 병렬 컴퓨터의 프로그래밍 모델은 우리가 익숙한 순차적 프로그래밍 모델과 다르다. 따라서 리눅스 클러스터를 가지고 '의미있는 작업'을 수행하기 위해서는 병렬 컴퓨터 아키텍처와 병렬 프로그래밍 모델에 대한 이해가 필요하다.

1 병렬 컴퓨터 아키텍처

병렬 컴퓨터는 2개 이상의 CPU로 구성되어 있으며, 여러 개의 CPU를 '동시에' 사용하면 주어진 일을 '빨리' 끝낼 수 있다는 생각으로 구성되는 아키텍처이다. 각각의 CPU가 독립된 메모리 주소와 프로그램 counter를 유지한다는 점에서 병렬 처리 컴퓨터(parallel processing computer)는 순차 처리 컴퓨터(serial processing computer)와 다르다. 여러 대의 컴퓨터가 사용자에게 어떻게 보여지는가에 따라서 병렬 컴퓨터 아키텍처를 여러 형태로 생각할 수 있다.

- **병렬화 정도에 따른 분류** : coarse grained vs. fine grained
- **Flynn's 분류** : SISD, SIMD, MIMD, MISD $\rightarrow$ SPMD
- **메모리 모델에 따른 분류** : distributed memory, shared memory,distributed-shared memory
- **노드간 물리적 연결 방법에 따른 분류** : switch, hardware supported remote memoery access by direct memory access, router chip(or processor) etc
- **노드간 연결의 논리적 구조 분류** : hypercube, 2-D mesh, 3-D torus, switch, etc
- single system image vs. multiple system image

하드웨어 구조의 분류와 상관없이 병렬 컴퓨터를 병렬로 사용하려면 병렬 프로그래밍을 해야 한다(어떤 프로그램이 병렬화가 가능한지 불가능한지를 컴파일러가 자동으로 결정할 수 없다).

기본적인 순차 처리 프로그래밍 construct(array, control structure, procedure, recursion)을 가진 여러 대의 컴퓨터를 대상으로 프로그래밍한다는 점에서 병렬 프로그래밍은 순차 프로그래밍과 다르다.

A man who has two women loses his soul. But a man who has two houses loses his head.

Leslie Lamport's example (cf. G.F. Pfister, "In search of clusters" 2nd ed., Prentice Hall, original ref. L. Lamport, IEEE Transactions on Computers, C-28(9), 690 (1979))

```
모두 x = y = 0 에서 시작 \\
/* Processor 1 */ \\
x = 1; \\
if (y == 0) print( 1 wins'' ); \\
/* Processor 2 */ \\
y = 1; \\
if (x == 0) print( 2 wins'' );
```

따라서 병렬 프로그램 작성자는 프로그램하려는 문제의 논리적 구조에 대해서 이해하고 있어야 한다.

병렬화 보기(쉬운 do-loop 구조) :

```
do i  = 1, MAX \\
  Z(i) = X(i) + Y(i) \\
enddo
```

• superscalar 아키텍처가 아닌 순차 컴퓨터의 경우

* 메모리로 부터 X(i)와 Y(i)를 regsiter로 읽어 들인다.
* X(i)와 Y(i)를 더한다.
* 더한 결과가 있는 register에서 Z 메모리 위치로 쓴다.
* 이 과정을 loop의 숫자만큼 반복한다.

• 병렬 컴퓨터의 경우 (위의 예제 : embarrasingly parallel 경우)

* MAX개의 X(i)와 Y(i)를 N대의 컴퓨터에 골고루 나누어(MAX/N) 분배
* MAX/N개의 X(i)와 Y(i)에 대하여 순차처리 컴퓨터의 과정을 N대의 컴퓨터가(MAX/N)-번 수행한다.
* 나누어 계산된 Z(i)를 한군데에 모은다.

• Amdahl's law : 병렬화에 따른 최대 성능 개선비

한 대의 컴퓨터상에서 코드 중 병렬화가 가능한 부분을 순차적으로 수행하는 데 걸리는 시간 = Ip
한 대의 컴퓨터상에서 코드 중 병렬화가 안되는 부분을 순차적으로 수행하는 데 걸리는 시간 = Is

$$\text{성능비} = \frac{Ip + Is}{\frac{Ip}{N} + Is}$$

예를 들어, Ip가 90%, Is가 10%의 CPU 시간을 차지한다면, 설사 N이 무한대라고 하더라도 성능 증가는 단지 (0.9 + 0.1)/(0.9/N + 0.1) = 10배에 불과하다.

물론 병렬화에 수반되는 통신에 의한 system overhead 때문에, 병렬화에 의한 전체 계산 속도는 Is가 0%인 경우에도 노드의 개수에 단순 비례하지 않는다. 따라서 계산을 수행함에 있어서 통신이 필요한 부분(surface)과 한 노드가 담당하는 부분(volume)과의 상대적인 비율을 낮추어야만 병렬 컴퓨터의 효율이 높다.

2 클러스터 아키텍처

병렬 컴퓨터의 대표적인 아키텍처는 MPP(Massively Parallel Processing) 컴퓨터와 SMP(Symmetric Multi-Processing) 컴퓨터이다 (CC-NUMA도 있지만). 클러스터 아키텍처는 MPP 아키텍처에 가깝지만, 전통적으로 병렬 컴퓨터의 구성 요소인 컴퓨터 노드가 한 대의 독립적인 컴퓨터로 작동할 수 있는 성질을 계속 유지하는 loosely-coupled 병렬 아키텍처이다. 따라서 분산 컴퓨터 아키텍처(distributed computer architecture)의 성질도 가진다.

클러스터 아키텍처에서는 독립적인 운영체제(OS)를 지니는 여러 대의 컴퓨터에 여러 개의 task를 효율적으로 분배하고 관리하는 것이 주된 관심사였다.(dynamic load balancing, high availability, etc)

따라서 병렬 컴퓨터와 클러스터 컴퓨터는 해결해야 하는 문제의 성격이 약간 달랐다. 병렬 컴퓨터에서는 통신에 따른 latency와 bandwidth 문제등의 해결에 많은 노력을 해왔지만 클러스터 컴퓨터에서는 job scheduling과 resource allocation 등과 같은 문제의 해결에 주력해왔다.

최근의 경향
- high performance computing
- high availiability system
- data base server
- web server
- more

3 리눅스 클러스터

리눅스 클러스터는 클러스터의 각 노드 운영체제로 리눅스를 사용하는 클러스터이다

Open Source인 리눅스 운영체제는 사용자가 off-the-shelf components를 가지고 직접 필요에 맞게 클러스터를 제작할 수 있게 한다.

리눅스 클러스터는 리눅스가 다양한 CPU와 device를 지원하기 때문에 클러스터 개념 자체를 쉽게 여러 가지 형태로 구현할 수 있고 리눅스의 발전에 따른 효과를 지속적으로 클러스터에 적용할 수 있는 '일반성' (generality)이 있다.

리눅스 클러스터는 사용자가 각자의 필요에 적합한 클러스터를 직접 구성하기 때문에 '효율성(efficiency)' 있다.
(리눅스의 일반성 때문에 효율성에 한계가 있기는 하다).

대량 생산되는 PC와 network 장비로 인한 저렴한 비용

리눅스 클러스터는 하드웨어에 상관하는 부분이 적어서 아키텍처 자체의 호환성(architecture portability)이 있다. 리눅스가 살아 있는한, 계속 발전되는 하드웨어의 사양(리눅스가 지원하는 사양)을 사용하여 클러스터를 '업그레이드' 할 수 있다.

de facto standard화한 message passing paradigm implementation에 따른 응용 프로그램의 호환성(software portability) 증가($\rightarrow$ MPI로 작성한 병렬 프로그램의 긴 수명 가능성)

확장성(scalability) 있는 아키텍처로 수 백대(Avalon의 경우 140대의 Alpha PC + Fast ethernet)로 확장이 가능하다.

직접 제작하는 리눅스 클러스터의 경우, 리눅스는 system 구성자에게 특수 하드웨어 개발과 이에 따른 운영체제와 compiler개발, hardware driver(network component의 software 측면) 개발과 유지, 보수에 관련된 부담을 덜어 주고 응용 프로그램 개발에만 치중할 수 있게 해 주었다.

2 리눅스 클러스터의 실제적인 구성

Premise : 하나의 병렬 컴퓨터 아키텍처가 모든 형태의 병렬 문제들에 적합한 것은 아니다. 클러스터 아키텍처도 예외는 아니어서, 한가지 형태의 클러스터가 모든 문제를 해결하는 데 효율적인 것은 아니다. 이러한 점에서 사용자가 손쉽게 직접 제작할 수 있는 리눅스 클러스터는 Open Source인 리눅스가 제공하는 다양한 노드 구조와 통신 관련 사항을 '사용자의 여건'에 맞게 선택할 수 있다. 물론 이 말은 제작하는 사람이 제작 목적을 정확히 이해하고 그 목적에 따라서 여러 가지 사양에 선택할 능력이 있어야 한다는 뜻이 될 수도 있다.

1 하드웨어 선택 사양

• 노드 CPU

* Intel vs. Alpha (or others)
* single CPU vs. SMP
* 병렬 컴퓨터는 동기화(synchronization) 때문에 가장 느린 노드 컴퓨터의 속도가 전체 속도를 결정한다

• 노드간 연결과 관련된 하드웨어 사항

* fast ethernet + 100 Mbps switch
* Myrinet NIC + Myrinet switch
* gigabit ethernet + gigabps switch
* SCI + SCI switch
* ATM card + ATM switch ???
* PLIP ???

• Hard disk system 선택

* big disk + diskless
* disk on every nodes
* local disk and i/o disk

• Serial console or not

* 소비 전력과 열
* 노드간 연결의 논리적 구조와 관련된 사항
* switch 구조의 경우 논리적 구조 선택이 상대적으로 자유롭다.
* multi-level switch, channel bonding ?
* multiple networks ?

• Hard disk system의 논리적 구조 선택

* share nothing (distributed file system)
* share something (예 : local partition + NFS partition)
* share everything (예 : diskless + NFS)

2 노드 네트워크 환경 구성

- single access point vs. multiple access points
- NIS
- rsh

3 병렬 환경 구성

기본적으로 리눅스 클러스터에서 현재 사용할 수 있는 병렬 환경은 메시지 패싱 방법이다.

* LAM (Local Area Multi-computer)
* MPICH (MPI-Chameleon)
* PVM (Parallel Virtual Machine)
* KMPI (Kool MPI)
* Setting-up

3 MPI를 사용한 리눅스 클래스터 프로그래밍

Premise : 리눅스 클러스터를 병렬로 사용하려면 병렬 프로그램을 사용해야 한다. 컴파일러가 순차 처리 프로그램을 자동적으로 병렬화 하지 않기 때문에, 이미 병렬로 작성된 응용 프로그램이 없다면 사용자가 직접 병렬 프로그램을 작성해야 한다. 병렬 프로그램을 작성하는 것은 순차 처리 프로그램 작성하는 것과 다른 프로그래밍 모델을 따르기 때문에 어려움이 따른다. 리눅스 클러스터에서 현재 사용 가능한 병렬 프로그래밍 환경은 message passing 형태이므로, 이 환경하에서 병렬 프로그램을 작성하는 것이 필요하다.

1 병렬 프로그래밍 방법

message passing : 정보가 local 메모리상에 존재하는 경우와 다른 컴퓨터의 메모리 상에 존재하는 경우로 구분하고, 병렬 프로그램상에서 non-local 정보가 필요할 때, message를 주고 받아서 해결한다.(분산 메모리 형태에 적합)

```
* 두 개의 노드 또는 다수의 노드가 협력한다.
* PVM (Parallel Virtual Machine)
* MPI (Message Passing Interface)
* BSP (Bulk Synchronous Parallel)
```

data parallel : 데이터의 구조를 미리 분배하는 형태로 선언한 후, 이렇게 선언한 정보가 local 정보인지 non-local 정보인지 필요에 따라서 컴파일러와 하드웨어가 해결한다.(공유 메모리 형태에 적합)

```
* 예를 들어 메모리에 대해서는 lock-unlock mechanism을 사용한다
* Open\_MP
* HPF
* F90

in-between
* Linda
```

2 MPI(Message Passing Interface)에 대한 간단한 소개

1994년 MPI forum이 MPI 표준을 정하기 전에는 다양한 아키텍처를 가진 여러 가지 병렬 컴퓨터를 위해서 프로그램을 작성하고 수행하려면 각각 다른 방법를 사용해야 했다. 이러한 문제는 병렬 컴퓨터 아키텍처의 장기적인 발전에 많은 지장을 초래한다고 예상되어, 병렬 컴퓨터와 관련된 많은 사람들이 forum을 결성하고 message passing paradigm을 이용하는 컴퓨터를 프로그램하고 사용하는 표준을 만들었다. 이 표준은 IEEE 같은 인증 기관이 인증한 표준은 아니지만, 슈퍼컴퓨터 industry의 de facto 표준의 역할을 하고 있어서 병렬 컴퓨터 하드웨어와 독립적으로 병렬 프로그램을 작성하고 유지할 수 있게 되었다.

이러한 시도는 상당히 성공적이어서, 공유 메모리 방식의 병렬 컴퓨터 아키텍처에서도 Open_MP라는 업계 공동의 표준을 1997년에 작성하는 계기가 되었다.

목적

- 메시지 패싱 형태의 병렬 프로그램 호환성 (portability)
- 효율성 (efficiency)은 계속 유지
- 자유로운 기능성 (functionality)

특징

- Message passing standard — portability
- based on previous message passing libraries
- Program interface
- Allow efficient communication
- Allow heterogeneous environment
- C and Fortran 77 binding — language independent semantics

3 기본적인 MPI Function(모두 127개의 함수)

message passing : 정보를 local 메모리 상에 존재하는 경우와 다른 컴퓨터의 메모리 상에 존재하는 경우로 구분하고, 병렬 프로그램 상에서 non-local 정보가 필요할 때, message를 주고 받아서 해결한다 (분산 메모리 형태에 적합).

• Task management (more)

- MPI_Init()
- MPI_Barrier()
- MPI_Wtime(), MPI_Wtick()
- MPI_Finalize()

• Communicator (more)

- MPI_Comm_create()
- MPI_Comm_rank()
- MPI_Comm_size()

• Topology management (more)

- MPI_Cart_create()
- MPI_Cart_coords()
- MPI_Cart_get()
- MPI_Cart_rank()

Communication (more)

- blocked send-receive : MPI_Send(), MPI_Recv(), MPI_Sendrecv(), etc
- non-blocked send-receive : MPI_Isend(), MPI_Irecv(), MPI_Wait()
- collective communication : MPI_Reduce(), MPI_Allreduce(), MPI_Bcast(), MPI_Scatter(), MPI_Gather()

• Derived Data type (more)

- MPI_Type_commit()
- MPI_Type_size()
- MPI_Type_struct()

Process group

4 병렬 프로그래밍 고려 사항

- load balance
- surface-to-volume ratio
- deadlock과 racing condition

5 MPI 프로그램 예제

기본적인 point-to-point communication 예(Appendix 1)
기본적인 collective communication과 operation 예(Appendix 2)
putting them all together
compile과 execution

● Appendix 1. 기본적인 point-to-point communication 예

```
#comm_mpi.c

#include <stdio.h>
#include </usr/local/lam-CFC/h/mpi.h>
#define MAX 500
main(int argc, char** argv)
{
  int i, node, numtask, nextup, nextdn, itag;
  float sendbuf[MAX], recvbuf[MAX];
  MPI_Status status;
  MPI_Request request;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&node);

  MPI_Comm_size(MPI_COMM_WORLD,&numtask);
  if (node != (numtask-1))
    nextup = node + 1;
  else
    nextup = 0;
      if (node != 0)
    nextdn = node - 1;
  else
```

```
    nextdn = numtask-1;
  for(i = 0; i < MAX;i++) {
    sendbuf[i] = node*0.1*(i+1);
    recvbuf[i] = 0.0;
  }
  itag = 22;
MPI_Irecv(&recvbuf,MAX,MPI_FLOAT,nextup,itag,MPI_COMM_WORLD,&request);
  MPI_Send(&sendbuf,MAX,MPI_FLOAT,nextdn,itag,MPI_COMM_WORLD);
  MPI_Wait(&request, &status);
  printf("from %d received %e\n", nextup, recvbuf[0]);
  MPI_Finalize();
}

#comm_mpi.f

      program comm_mpi
      parameter(MAX=500)
      include '/usr/local/lam-CFC/h/mpif.h'
      integer istatus(MPI_STATUS_SIZE)
      real sendbuf(MAX), recvbuf(MAX)
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,node,ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,numtask,ierr)
      if (node .ne. (numtask-1)) then
        nextup = node + 1
      else
        nextup = 0
      endif

      if (node .ne. 0) then
        nextdn = node - 1
      else
        nextdn = numtask-1
      endif
      do 10 i = 1, MAX
        sendbuf(i) = node*0.1*i
        recvbuf(i) = 0.0
 10   continue
      itag = 22
      call MPI_IRECV(recvbuf,MAX,MPI_REAL,nextup,itag,
     #     MPI_COMM_WORLD,ireq,ierr)

      call MPI_SEND(sendbuf,MAX,MPI_REAL,nextdn,itag,
     #     MPI_COMM_WORLD,ierr)
      call MPI_WAIT(ireq,istatus,ierr2)
      write(6,*) 'from ', nextup, ' receive ', recvbuf(1)
      call MPI_FINALIZE(ierr)
      stop
      end
```

Appendix 2. 기본적인 collective communication과 operation 예

#reduce_mpi.c

```
#include <stdio.h>
#include </usr/local/lam-CFC/h/mpi.h>
#define N 1000000
main(int argc, char** argv) {
int   i, node, numtask, subN, iroot;
float A[N];
double sum, sumt;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&node);
MPI_Comm_size(MPI_COMM_WORLD,&numtask);
subN = N/numtask;
for(i = 0;i < subN;i++) A[i] = (float)(i+1 + node*subN);
sum = 0.0;
sumt = 0.0;
iroot = 0;
for(i = 0;i < subN;i++) sum = sum + A[i];
MPI_Reduce(&sum,&sumt,1,MPI_DOUBLE,MPI_SUM,iroot,MPI_COMM_WORLD);
printf("total sum = %f\n", sumt);
MPI_Finalize();
}
```

#reduce_mpi.f

```
      program test_reduce
      include '/usr/local/lam-CFC/h/mpif.h'
      parameter (N=1000000)
      real A(N)
      double precision sum, sumt
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,node,ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,numtask,ierr)
      isubloop = N/numtask
      do i = 1, isubloop
        A(i) = real(i +  node*isubloop)
      enddo
      sum = 0.0
      sumt = 0.0
      iroot = 0
      do i = 1, isubloop
        sum = sum + A(i)
      enddo
      call MPI_REDUCE(sum,sumt,1,MPI_DOUBLE_PRECISION,MPI_SUM,iroot,
     &     MPI_COMM_WORLD,ierr)
      write(6,*) 'total sum = ', sumt
      call MPI_FINALIZE(ierr)
      stop
      end
```