Paralel Hesaplama Kütüphaneleri (Mpi, Pvm) ve Linux Ağında Çalıştırılmaları

Abbas Ayhan Kanmaz

İstanbul Teknik Üniversitesi
Bilişim Enstitüsü
80626, Maslak, İstanbul
kanmaz@be.itu.edu.tr

Anahtar Kelimeler: Dağıtık Hesaplama, Paralel Hesaplama, Kümeleme, Mpi , Pvm

Özet Bu çalışmada, çok yüksek hesap gücü gerektiren problemlerin çok işlemcili çok pahalı bir makinada çalıştırılmasındansa, aynı anda bir ağ ortamında, birçok makinada çalıştırabileceği anlatılmaktadır. Burada, bu amaçla tasarlanmış Mpi ve Pvm kütüphaneleri ile yazılmış programların bir ağ ortamında nasıl çalıştıracağı anlatılmıştır . Performans değerlendirilmesi yapılmamıştır. Performans değerlendirmelesi isteyenler bu konferanstaki 49 numaralı sunuma incelenebilir.

1 Paralel ve Dağıtık Hesaplama

Teknoloji geliştikçe, insan hayatında karşılaşılan problemlerin karmaşıklığı da artmaktadır. Bilimadamları, karşılaştıkları matematik problemlerine önce analitik çözüm ararlar. Ancak problemlerin analitik olarak çözülmesi, karmaşıklıkları yüzünden her zaman mümkün değildir. Bu amaçla, araştırmacılar bilgisayarların yardımına başvururlar ve problemlere sayısal çözümler ararlar.

Bilgisayarların işlem gücü çok fazladır ve çok az zamanda çok fazla matematik işlemi yapabilirler. Ancak teknoloji ve bilim ile birlikte problemlerin zorluk dereceleri de artar ve bazen tek bir bilgisayarın bir problem çözümü için saatlerce bazen de aylarca çalışması gereklidir. Tabi ki, aylar sonra öğrenebileceğimiz bir cevap bizim işimize yaramayabilir. Bu nedenle, bazen araştırmacılar programlarını gözden geçirerek çeşitli optimizasyonlara (özellikle döngü ve dizilerde) gidebilirler. Bu durumda kazanılan zaman bazen yeterli olabilir. Ancak yeterli olmadığında programı daha hızlı çalıştırmak için derleyici parametreleri kullanılarak hız kazanılmaya çalışılabilir. Bu işlemde her zaman istediğimiz zamanı bize kazandırmayabilir.

Çok büyük bir problemi daha az zamanda çözmenin başka yolları da vardır. Bu amaçla problemin doğası incelenebilir. Eğer probleminiz parçalanmaya elverişli ise ve sizinde birden fazla bilgisayarınız veya birden fazla işlemciniz varsa, herbir parçayı farklı bir bilgisayar veya işlemcide çalıştırabilirsiniz.

Bu amaçla tasarlanmış çok işlemci ve çok büyük hesaplama yetenekleri olan bilgisayarlara süper bilgisayarlar denir. Süper bilgisayarların işlemci güçleri, bel-

lekleri ve depolama kapasiteleri çok yüksektir. Ancak bu kadar güçlü özellikleri nedeniyle çok yüksek maliyetlidirler.

Eğer elimizde bir çok işlemci bir makina yok ise ve maliyeti nedeniyle bir süper bilgisayar üzerinde çalışamıyorsak, problemimizi kendi ağ ortamında da çözebilme şansımız vardır. Bu durumda ağdaki bütün bilgisayarları bir bilgisayarmış gibi düşünüp bütün bilgisayarları da birer işlemci gibi düşünebiliriz. Ancak bu durumda, eğer program parçaları birbirleri ile çok fazla haberleşiyorsa, bunun hızı anakartın hızında değil de ağ hızında olacaktır.

2 Paralel Programlama Modelleri

Programın nasıl bölüneceği ve her işlemcide nasıl çalışacağı da çok önemlidir. Burada en önemli konu, programı elinizdeki işlemci gücüne göre bölmeniz gerektiğidir. Öneğin bazı işlemcilerinizin gücü diğerlerine göre çok fazla veya az olabilir. Eğer siz bunlara, diğerlerine gönderdiğiniz kadar işlem gönderirseniz. Programın sonlanması için az kapasiteli işlemcilerinizin işlerini bitirmesini beklemeniz gerekir. Burada amaç bütün işlemcilerinizin en az zamanda beklemesidir.

Program parçalarının nasıl çalışacağı da önemli bir konudur. Program parçaları, iki farklı şekilde olabilir. Bazen program parçaları thread şeklindedir. Bu durumda işlemci sayısı kadar thread vardır ve belleğiniz ortak paylaşımlı bellektir (shared memory). Program parçalarının bu şekilde olması için bir süper bilgisayara veya çok güçlü bilgisayara ihtiyacanız vardır. Bu çözümün avantajı, programlamasının çok kolay olmasıdır. Belirli sayıda işlemci için çok güzel sonuçlar alırsınız. İşlemci sayınızı artırdığınız oranda hız düşer. Ancak, kullandığınız bilgisayar ortak paylaşımlı olduğundan , işlemcili sayısını çok çok artırdığınızda belleğin paylaşılmasından dolayı sıkıntılar yaşarsanız ve işlemci sayısınızı artırdığınızda hızınız azalmaz, aksine artabilir. Yani, bu programlama modeli çok fazla işlemci için ölçeklenebilir değildir.

Bir başka çözümde program parçalarının herbirini bir süreç olarak çalıştırmaktır. Bu durumda her işlemci veya bilgisayar bir süreç alıp çalıştırır. Diğerine göre bu tarz programlama tekniği çok daha zordur. Bu durumda, program ölçeklenebilir ancak, burada da süreçlerin birbirleri ile haberleşme zamanı önemlidir. Eğer farklı bilgisayar kullanıyorsanız bazen programı belirli sayıdan fazla parçaya bölmek iyi bir çözüm değildir. Ancak problemin boyu büyümüş ise, bu sayıyı artırdığınızda yine hız elde edebilirsiniz. Yani bu durumda düşünmeniz gereken şey haberleşme ve hesaplama zamanlarının karşılaştırılmasıdır. Ağ ortamındaki bilgisayarlar, bu tip programlama tekniğini süreçleri birbirlerine göndererek yapabilirler. Bunun için kullanılan program kütüphanelerine mesaj geçme arayüzü (message passing interface) denilir.

3 Mesaj Geçme Arayüzü

Mesaj geçme arayüzü, bir programın birden çok sürece bölünerek her bir sürecin bir bilgisayarda veya işlemcide çalıştırılmasıdır. Bu tarz programlamada bilinen,

standart C,Fortran,C++, vb... programlama dilleri kullanılabilir. Burada mesaj geçme arayüzünü sağlayan programlama kütüphaneleridir.

En fazla bilinen, mesaj geçme arayüzü kütüphaneleri şunlardır

* Mpi (Message Passing Interface)
* Pvm (Parallel Virtual Machine)

Bunların avantajı hem bir süperbilgisayarda, hem de ağ ortamında, birçok farklı özellikteki bilgisayarda çalışabiliyor olmasıdır. Böylelikle programın taşınabilirliği sağlanır.

Aslında her iki uygulamada tek bir yerel bilgisayara kurularak da çalışılabilir. Çünkü mesaj geçme arayüzlerinin mantığı süreç oluşturulması üzerine kuruludur. Eğer elinizde işlemci sayınızdan fazla süreç varsa, kalan süreçler yine işlemciler arası paylaştırılır.

Günümüzde Mpi daha fazla yaygın olarak kullanılmakta olan bir defacto standartdır. Pvm ise daha çok dağıtık hesaplama ve ağ ortamı düşünülerek hazırlanmıştır. Pvm, Ulusal Oakridge Laboratuvarlarında geliştirilmiştir. Bir diğer kütüphane olan Mpi ise değişik uygulamaları sahiptir. Bunlardan en çok bilinen 3 tanesi şunlardır:

* Mpich : Ulusal Argonne Laboratuvarı
* Lam : Ohio Super Bilgisayar Merkezi
* Chimp : Edinburg Paralel Hesaplama Merkezi

4 Linux Ağı üzerinde Paralel Hesaplama Ortamının Hazırlanması

Bu calışma, zaten işleyen bir ağ için düşünülmüştür. Yani, herhangi bir Linux öğrenci Laboratuvarı iyi bir ortam olabilir. Eğer çok Bir diğer kütüphane olan hesaplama gücüne ihtiyacınız varsa, tabi ki ağ ortamındaki bilgisayaların başkaları tarafından meşgul edilmesi sizin işinizi yavaşlatacaktır. Bu durumda hesaplama işleriniz için ayrı bir ağ oluşturabilirsiniz. Bu durumda çekirdeği buna göre hazırlanmış bir Linux kullanmak size daha hızlı hesap yapmanız için yardımcı olacaktır. Ancak bu durumda bilgisayarları başka işler için çalıştırmak mümkün olmayacaktır. Böyle bir durumda ihtiyaçlar ve kaynaklar iyice analiz edilmelidir.

Bir ağı paralel hesaplamada kullanmak için öncelikle dns servisi kurmak, işleri rahatlatıcaktır. Böylelikle Ip adresleri yerine isimler kullanılarak işler daha kolay çözümlenir. Daha sonra ise, eğer sağlıklı bir ağ ortamı varsa nfs+nis ikilisi işleri çok rahatlatacaktır. Böylelikle, derlenmiş programlar aynı kullanıcı adı için ağdaki bütün bilgisayarlarda paylaştırılmış olur. Aksi takdirde programın derlenmiş şekli teker teker bilgisayarlara kopyalanmalıdır. İstenildiği takdirde, bütün derleyiciler, yorumlayıcılar ve kütüphaneler de, bir bilgisayara konulup teker teker kurulum yapma işlemi engellenmiş olur.

Eğer bütün bunlar hazırsa bilgisayarlar arası iletişimin hangi servis ile olucağı seçilebilir. Mesaj geçme arayüzü kullanarak yazılan programlarda program bir bilgisayarda çalıştırıldığında, süreçler diğer bilgisayarlara dağılır. Ancak bu dağılma işlemi sırasında, programın diğer bilgisayarda çalışması için kullanıcı adı ve

şifresi sorulmadan diğer bilgisayara geçmesi gerekir. Bu amaçla iki servis yaygın olarak kullanılır.

* rsh : Uzak erişim kabuk (remote shell)
* ssh : Güvenli kabuk (secure shell)

4.1 Uzak Erişim Kabuk(rsh)

Bir kullanıcı,bu kabuk yardımıyla ev dizinindeki .rhosts dosyasina hangi makinadan gelen hangi kullanıcının kendi hesabına kullanıcı adı ve şifresi sorulmadan yazar. Dosyanın ($HOME/.rhosts) genel sözdizimi şöyledir:

```
bilgisayar kullanici_adi
```

Burada kayıtlar alt alta eklenebilir. Eğer burada kullanıcı adı kısmı yazılmaz ise ancak dışarıdan gelen kullanıcı ile giriş yapan aynı kullanıcı ise sisteme kullanıcı adı ve şifresi sorulmadan giriş yapar.

Rahat ve kolay bir kullanıma sahip olmasına rağmen rsh seçilmesi çok sağlıklı değildir. Örneğin böyle bir uygulama, öğrenci laboratuvarında yapılıyor ise çeşitli güvenlik sorunları ile karşılaşıbilinir. Eğer paralel hesaplama kütüphaneleri güvenlik sorunları yaşanabilecek bir yerde çalıştırılacaksa rsh yerine ssh (güvenli kabuk) tercih edilmesi faydalıdır.

4.2 Güvenli Kabuk(ssh)

Parallel hesaplama kütüphanelerini kullanırken programımızın süreçlerinin diğer bilgisayar veya bilgisayarlara kullanıcı adı ve şifresi sorulmadan geçmesi gerekmektedir. Bu nedenle, güvenli kabuğun bazı özellikleri kullanılabilir. İlk öncellikle bir anahtar üretmek gerekir. Daha sonra bu anahtara sahip girişlerin güvenli olduğunu anlatmak gerekir. Bunları bir kereliğine yapmak yeterlidir. Anahtar rsa veya dsa algoritmalarına göre üretilmiş olabilir. Burada kullanılan güvenli kabuğun hangi sürüm (1 veya 2) çalıştığıda önemlidir. Alttaki örnekte, dsa algoritmasına göre üretilmiş ve 2. sürüm çalışan bilgisayarlarda bunun güvenli olduğu anlatılmıştr.

```
ayhan@gilda:~> ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/ayhan/.ssh/id_dsa):
Created directory '/home/ayhan/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ayhan/.ssh/id_dsa.
Your public key has been saved in /home/ayhan/.ssh/id_dsa.pub.
The key fingerprint is:
2a:5t:d4:60:0d:cd:12:38:af:05:41:54:61:22:6f:4f ayhan@gilda
ayhan@gilda:~> cp ~/.ssh/id_dsa.pub ~/.ssh/authorized_keys2
```

Bu işlem sadece bir kereliğine yapıldıktan sonra her paralel çalışacak programı çalıştırmadan yapılacak olan şu işlemleri yapmak gerekir.

```
ayhan@gilda:~> ssh-agent bash
ayhan@gilda:~> ssh-add .ssh/id_dsa
Identity added: .ssh/id_dsa (.ssh/id_dsa)
```

Bundan sonra güvenli kabuk ile kullanılacak bilgisayarlara birer kere uzaktan erişilip çıkılır. Burada ssh-agent programı özel üretilmiş anahtarınızı saklamak için kullanılır. Ancak kendisinin özel bir anahtarı olmadığı için ssh-add komutu ile anahtar eklenebilir. Bilgisayardan çıktığınızda ssh-agent komutunun çalışması bittiğinden her kullanım için bunu tekrar oluşturmanız gerekir.

Buradaki anlatım kullanıcıların disk alanlarına, ağdaki bütün bilgisayarlardan ulaşabildikleri varsayımı ile yapılmıştır. Ancak, böyle bir durum yoksa, bir bilgisayarda ürettiğiniz anahtarı diğerinde güvenli anahtar olarak kopyalanmanız gerekir.

5 Linux Üzerinde Mpi

Linux üzerinde Mpi kullabileceğiniz birçok uyugulama programı vardır. Bunların en fazla bilineni Mpich 'dir. Burada da, Mpich ten bahsedilecektir.

5.1 MPI Kurulumu, Derleme ve Çalştırılması

Mpich uygulaması, Ulusal Argonne Laboratuvarının adresinden indirilebilir. Aynı zamanda birçok Linux dağıtımı için rpm paketleri mevcuttur. Ayrıca kaynak kod autoconf paketiyle hazırlandığı için kurulum çok basittir. Burada dikkat edilmesi gereken nokta, configure ekleyeceğiniz ekstra parametlerdir. Bu komuta parametre girerek, standart olarak kullanılmayan derleyicilerinizide Mpi ile kullabilirsiniz. Mpi ile kullanılan en önemli parametreler makina mimarisini ve çalışma tipini söylediklerinizdir. Biz Linux' u bir ağ ortamında kullanacağımız için bunlar LINUX (ancak değişik Linux makinalar için değişebilir) ve ch_p4 şeklinde olmalıdır. Eğer güvenli kabuk ile çalışmak istiyorsanız kurulum öncesi ;

```
gilda:/tmp/mpich-1.2.2.3 # RSHCOMMAND=ssh
gilda:/tmp/mpich-1.2.2.3 # export RSHCOMMAND
```

şeklinde (csh ve türevlerini kullanıyorsanız setenv kullanmanız gerekir.) bir komut yürütmeniz gerekir.

Daha sonra ise kurulum yaptığınız dizine gidip "share/machine.LINUX" dosyasında çalışacağınız makinaları adlarını sırasıyla yazmanız gerekir. Bu dosya önemlidir, eğer bu dosyaya yazdığınız makinalardan birisi kapalı olursa ve programınız o makinada takılacaktır. Bunu önlemek için mpirun komutu ile beraber "-machinefile" parametresi kullanlabilir.

Kurulum sonrası Mpi kütüphanesi kullananan programlarınızı derleyebilirsiniz. Ancak kurulum yaptığınız dizindeki "bin/" dizinini $PATH değişkeni içerisine almanız iyi olacaktır. Derleme işleminde genel olarak;

* mpicc : C programlarını derlemek için kullanabilirsiniz.
* mpiCC : C++ programlarını derlemek için kullanabilirsiniz.
* mpif77 : f77 programlarını derlemek için kullanabilirsiniz.
* mpif90 : f90 programlarını derlemek için kullanabilirsiniz (Eğer f90 derleyiciniz varsa ve bunu kurulumda belirtmiş iseniz).

Örnek bir derleme aşağıda gösterilmiştir.

```
ayhan@gilda:~> ls cpi
ls: cpi: Böyle bir dosya ya da dizin yok
ayhan@gilda:~> mpicc -o cpi cpi.c
ayhan@gilda:~> ls cpi
cpi
```

Bütün derleme işlemlerini yaptıktan sonra , programınızı çalıştırabilirsiniz. Bunun için kullandığınız kabuk ile ilgili, eğer önceden yapılması gereken bir işlem varsa yaptıktan sonra mpirun komutu ile programı çalıştırabilirsiniz. Bu komutu çalıtırırken "-np" parametresinden verdiğiniz sayı, yaratılacak süreç sayısıdır. Burada süreç sayısının elinizdeki işlemci sayısı kadar olması mantıklıdır. Bilgisayar ağlarında önemli olabilecek diğer bir parametrede "-machinefile" parametresidir. Bu parametre ile kullanıcı kendi çalışmak istediği makinaları seçebilir. Böylelikle machine.LINUX dosyasında yazılı bir makina kapalı olsa bile çalışmaya devam edilebilir. Örnek bir mpi programı çalıştırılması aşağıda verilmiştir;

```
ayhan@gilda:~> cat makina.mpi
gilda
terminator
alan
ayhan@gilda:~> mpirun -machinefile makina.mpi -np 3 cpi
```

5.2 Basit Mpi Fonksiyonları

Mpi daha öncede belirtildiği gibi paralel hesaplamalar için üretilmiş bir kütüphanedir. Sonuçta kütüphane olduğuna göre mpi kullanırken mutlaka bir programlama dili yardmıyla bu kütüphaneyi kullanırız.Mpi sayısal hesaplamalarda yaygın olarak kullanılan C,Fortran gibi programlama dilleri yardımıyla kullanılabilir. Hatta perl üzerinde Mpi ile çalışmak için bir modülde bulunmaktadır. Herhangi bir programlama dilinde bu kütüphaneyi kullanmak için tabiki, kütüphanenin adının program başlarken belirtilmesi gerekir. Böylelikle derleyici ya da yorumlayıcıya o kütüphanenin fonksiyonlarının kullanılacağı söylenmiş olunur (C için mpi.h, fortran icin mpif.h).

Mpi programı yazılırken izlenen yol genellikle, bir süreci ana süreç olarak kabul edip diğer süreçlerin işçi süreçler olması ve bütün işlemlerin tek bir programda yazılması şeklindedir.

Çok basit olarak Mpi kullanmak için, ilk öğrenilmesi gereken en temel Mpi fonksiyonları ;

* MPI_Init: Programın bu kısmından sonrası "-np" parametresi ile verilen süreç sayısı için yapılır..
* MPI_Finalize: Programın bu kısmından sonra programın çok süreç şeklinde çalışması duracaktır.
* MPI_Comm_rank: Bu fonksiyon sayesinde "-np" parametresi ile verilen süreçler, tek olarak 0 ile n-1 arası numara alırlar. Böylelikle, her sürecin değişik numarası olur ve program içinde bu değerler kullanılarak değişik süreçlere değişik işler yaptırılabilir.
* MPI_Comm_size: Bu fonksiyon sayesinde süreçler toplam süreç sayısını öğrenirler:
* MPI_Send: Bir sürecin bir diğerine değişken göndermek istediği zaman bu fonksiyon kullanılır. Tek bir değişken gönderilebildiği gibi tamamen bir dizi veya bir dizinin bir kısmıda bir kerede gönderilebilir.
* MPI_Recv: Bir süreç, başka bir sürecin gönderdiği değişkeni veya değişkenleri bu fonksiyon yardımıyla alır.

Genelikle kullanılan diğer Mpi fonksiyonları ise değiken gönderme ve alma üzerine değişik açılımlar olan fonksiyonlardır. Mpi kütüphanesi kullanarak yazılmış bir C programına örnek verilmiştir.

```
* Program Peter S. Pacheco  */
/* A User's Guide to MPI     */
/* makalesinden uyarlanmistir*/
#include <stdio.h>
#include "mpi.h"
int main(int argc, char** argv) {
    int      surecno;  /* Süreç numarası     */
    int      boyut;    /* Toplam Süreç Sayısı*/
    double   a = 0.0;  /* Integral Başlangıcı*/
    double   b = 1.0;  /* Integral Bitişi    */
    int      n = 1024; /* Aralık miktarı     */
    double   h;        /* Aralık yüksekliği  */
    double   yerel_a;  /* Süreç için baş.    */
    double   yerel_b;  /* Süreç için bitiş   */
    int      yerel_n;  /* Her sürecin için   */
                       /* aralık  sayısı     */
    double   integral; /* Sürecin aralığında */
                       /*  Integral          */
    double   toplam;   /* Toplam Integral    */
    int      kaynak;   /* Kaynak Süreç       */
    int      hedef = 0;/* Hedef              */
    int      tag = 0;  /* Tag numarası       */
    MPI_Status  status;
    double Trap(double yerel_a, double yerel_b,
int yerel_n, double h);
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &surecno);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &boyut);
    h = (b-a)/n;
    yerel_n = n/boyut;
    /* Bütün süreçler aşağıdaki aralık*/
    /* için çözüm yapacaklardır       */
    /* integration = yerel_n*h.       */
    yerel_a = a + surecno*yerel_n*h;
    yerel_b = yerel_a + yerel_n*h;
    integral = Trap(yerel_a, yerel_b, yerel_n, h);
    if (surecno == hedef) {
        toplam = integral;
        for (kaynak = 1; kaynak < boyut; kaynak++){
            MPI_Recv(&integral, 1, MPI_DOUBLE, kaynak,
tag, MPI_COMM_WORLD, &status);
            toplam += integral;
        }
    } else {
        MPI_Send(&integral, 1, MPI_DOUBLE, hedef,
            tag, MPI_COMM_WORLD);
    }
    if (surecno == hedef) {
        printf("%4d aralik sayisi ile \n", n);
        printf("integral(%6.2lf,%6.2lf) aralığında
= %11.5lf olarak bulundu\n",
            a, b, toplam);
    }
    MPI_Finalize();
}
double Trap(
          double  yerel_a,
          double  yerel_b,
          int     yerel_n,
          double  h){
    double integral;
    double x;
    int i;
    double f(double x);
    integral = (f(yerel_a) + f(yerel_b))/2.0;
    x = yerel_a;
    for (i = 1; i <= yerel_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
```

```
}
double f(double x) {
    double donus_deg;
    donus_deg = 5*x*x*x*x;
    return donus_deg;
}
```

6 Linux üzerinde Pvm

Pvm' de Mpi gibi, genellikle bir ana süreç ve onun kontrolündeki işçi süreçler kullanılarak yapılan bir programlama tekniğine göre çalışır. Ancak buradaki fark, akside mümkün olsa bile Pvm' de 2 ayrı program yazılır. Bunların birinci ana sürecin çalıştırdığı programdır. Diğer programı ise işçi süreçler çalıştırır.

Pvm , bir ağda değişik mimaride bilgisayar arasında çalışabilecek şekilde tasarlanmıştır. Değişik bilgisayarları tek bilgisayardaki çok işlemci var gibi kabul ederek öyle çalışır. Bunun için bir sanal makina kullanır. Bu nedenle program çalıştırılmadan önce sanal makinaya bilgisayarlar eklenir ve böylelikle bir işlemci havuzu elde edilmiş olur.

Pvm' in en büyük farkı, açılacak süreç sayısı ana süreç için yazılan programın içinde tanımlanır. Yani program çalışmadan önce süreç sayısı bir argüman olarak girilmez.

6.1 Pvm Kurulum, Derleme, Çalıştırılması

Pvm içinde kendi adresinden kaynak dosyalar indirilip kurulum yapılabilir. Mpi' da olduğu gibi değişik Linux sürümleri için rpm paketleri bulunabilir.

Kaynak kod derlenmeden önce Pvm kurulacak dizini PVM_ROOT çevre değişkeni olarak belirlemek gerekir. Daha sonra, bu değişkeni kullanıcıların başlangıç betiklerine yazmaları iyi olur. Bundan başka PVM_ARCH değişkeni de vardır. Bu değişkende makina mimarisini belirtir. Makina mimarisi;

```
$PVM_ROOT/lib/pvmgetarh
```

betiği yardımıyla öğrenilebilir. Bu iki değişkenden belirtikten sonra, eğer güvenli kabuk kullanılmak istenirse;

```
$PVM_ROOT/conf/$PVM_ARCH.def
```

dosyasinin içindeki "/usr/bin/rsh", "/usr/bin/ssh" (veya güvenli kabuk nerede kurulu ise) olarak değiştirilmelidir. Ama, bu yapılmadığı zamanda güvenli kabuk kullanmak için şans vardır.

Bütün işlemlerden sonra, derlemenin tamamlanası için make yazmanız yeterlidir. Eğer ekstra programlarıda kurmak isterseniz make all yazmanız yeterli olacaktır. Bundan sonraki kullanımlar için $PATH çevre değişkenini tanımlamanız yerinde olacaktır. Bunları tanımlamak için örnek dosyaları;

```
$PVM_ROOT/lib/
```

dizini içerisinde, adı kullandığınız kabuk, uzanıtısı ise stub olarak bulabilirsiniz.

Pvm kütüphanesini kullanan programları derlemek, programı kurma şekline göre değişebilir. Eğer kaynak koddan derleme yapmış iseniz, aşağıdaki gibi bir derleme işinize yarabilir;

```
c için;
ayhan@gilda:~> gcc  -I$PVM_ROOT/include -o hello hello.c \
 -L$PVM_ROOT/lib/$PVM_ARCH -lgpvm3 -lpvm3
f77 için ise;
ayhan@gilda:~> g77  -I$PVM_ROOT/include -o master1 master1.f \
 -L$PVM_ROOT/lib/$PVM_ARCH -lfpvm3 -lpvm3
```

Görüldüğü gibi aslında yapılan şey, ilk olarak kütüphane dosyasının yerini söylemek ve daha sonra ise kütüphanelere gerekli bağlantıların yapılmasıdır. Derleme işlemi için "aimk" komutu yardımıyla yazılacak olan Makefile' dan da yararlanılabilir.

Derleme sonrası yapılması gereken işlem sanal makinanın çalıştırılmasıdır. Bu amaçla, önce bağlantı sorunları çözülmeli. Eğer güvenli kabuk kullanılacaksa bağlantı için daha önceden belirtilen gerekli işlemlerin yapılmalıdır. Eğer Pvm güvenli kabuk destekli kurulmadıysa ve güvenli kabukda çalışmak isterseniz;

```
ayhan@gilda:~> export PVM_RSH=/usr/bin/ssh
```

şeklinde bir komut yürütülerek, kullanılacak kabuk güvenli kabuk olarak seçilebilir. Daha sonra ise, güvenli kabuk için daha önce belirtilmiş olan işlemlerin yapılması gerekir.

6.2 Pvm Konsolu

Pvm sanal makinasını oluşturmak için yardımcı bir programa ihtiyaç duyulur. Bu amaçla pvm komutu kullanılır. Bu komut size ayrı bir kabuk açarak, çalışacağınız makinalari kendi işlemci havuzunuza eklemeye veya çıkarmaya yarayacaktır. Bu amaçla, pvm yazdığınızda çalıştığınız makinada bir servis başlar ve siz makinaları ekledikçe , eklediğiniz makinalarda bu servis çalıştırılır. Bu konsol kullanmak için kullanabileceğiniz bazı komutlar şunlardır:

- **help:** Bu komutu konsolda yardım almak için kullabilirsiniz.
- **add:** Sanal makinanıza makina eklemek için kullanılır.
- **delete:** Sanal makinanızdan makina çıkarmak için kullanılır.
- **conf:** Sanal makinanizdaki makinalari görmek için kullanılır
- **ps:** Çalışan görevleri listeler.
- **reset:** Bütün görevleri öldürür.
- **quit:** Pvm konsolundan çıkmak için kullanılır.
- **halt:** Her bilgisayardaki Pvm servislerini durdurur ve konsoldan çıkar.

Örnek konsol kullanımı alttaki gibidir.

```
user@ayasofya:~ > pvm
pvm>conf ( to see the configuration )
conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
                 ayasofya    40000    LINUX    1000 0x00408841
pvm>add tas
add tas
1 successful
                    HOST     DTID
                     tas    80000
pvm> add gokova
add gokova
1 successful
                    HOST     DTID
                   gokova   100000
pvm> conf
conf
3 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
                 ayasofya    40000    LINUX    1000 0x00408841
                      tas    80000    LINUX    1000 0x00408841

pvm> delete tas
delete tas
1 successful
                    HOST  STATUS
                     tas  deleted
pvm> conf
conf
2 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED       DSIG
                 ayasofya    40000    LINUX    1000 0x00408841
                   gokova   100000    LINUX    1000 0x00408841
```

Burada dikkat edilmesi gereken şey quit komutu sadece sizi konsoldan çıkarır. Ancak pvm servisleriniz, çalışmaya devam eder. Bunları kapatmak için halt komutu kullanılmalıdır. Sanal makinanizida çalışır duruma getirdikten sonra programı çalıştırabilirsiniz. Programı çalıştırdıktan sonra halt ile Pvm servislerini kapatmayı unutmayınız.

6.3 Basit Pvm Fonksiyonları

Aynı Mpi'da olduğu gibi Pvm'de de genellikle süreçler birbirleri ile haberleşirler. Pvm kütüphanesindeki en büyük fark süreçlerin program içinden açılmasıdır. Pvm tarzı programlamada genellikle, değişkenler gönderilmeden önce

paketlenir. Değişkeni alan süreç, bu paketi önce açıp daha sonra kullanabilir. En çok kullanılan bazı Pvm fonksiyonları şunlardır:

* pvm_spawn: Ana süreç içinde istenilen sayıda işçi süreç açmak için kullanılır. Çok sürece geçişin başlangıcıdır.
* pvm_exit: Pvm' den çıkmak için kullanılır.
* pvm_mytid: Her süreç, bu fonksiyon yardımı ile kendi kimliğini öğrenir.
* pvm_parent: Her işçi süreç, bu fonksiyon yardımı ile ana sürecin kimliğini öğrenirler.
* pvm_pk...: Değişkenleri paket yapmak için kullanılır. Burada pk'dan sonra gelecek şey paketlenecek değişkenin tipine (tam sayı (int), kayar noktalı sayı (float), katar (string), vb ...) göre değişir.
* pvm_upk...: Başka bir düreç tarafından gönderilen değişkenlerin paketlerini açmak için kullanılır. Burada upk'dan sonra gelecek şey paketlenecek değişkenin tipine (tam sayı (int), kayar noktlı sayı (float), katar (string), vb ...) göre değişir.
* pvm_send: Paketlenmiş değişkenleri başka bir sürece göndermek için kullanılır.
* pvm_recv: Başka bir süreçten elen değişkenleri almak için kullanılır.

Örnek bir C programı aşağıda verilmiştir.

```
ana program:
/* PVM kaynak kodundaki */
/* hello.c orneginden   */
/* uyarlanmistir.       */
#include <stdio.h>
#include "pvm3.h"

main()
{
int cc, tid;
char buf[100];

printf("Benim kimligim %x\n",
 pvm_mytid());

cc = pvm_spawn("./merhaba_digeri",
 (char**)0, 0, "", 1, &tid);

if (cc == 1) {
cc = pvm_recv(-1, -1);
pvm_bufinfo(cc, (int*)0,
(int*)0, &tid);
pvm_upkstr(buf);
printf("%x gelen mesaj: %s\n",
tid, buf);
```

```
} else
printf(" merhaba_digeri
calistirilamiyor.\n");

pvm_exit();
exit(0);
}
isci program:
/* merhaba_digeri.c */
#include "pvm3.h"

main()
{
int ptid;
char buf[100];

ptid = pvm_parent();

strcpy(buf, "Merhaba ben ");
gethostname(buf + strlen(buf),
64);
pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, 1);

pvm_exit();
exit(0);
}
```

7 Karşılaşılabilecek sorunlar

Her iki kütüphane içinde en önemli sorun makinalar arası bağlantı sorunudur. Eğer programınız herhangi bir yerde takılıyorsa, makinalarınıza kullandığınız kabuk ile bağlanmayı deneyin. Eğer sorun burada değilse kontrol etmeniz gereken en önemli şey gerekli çevre değişkenlerini atayıp atamadığınızdır. Bunları kontrol ediniz. Bunların yanında dns, nfs, nis ayarlarınıza ve çalışıp çalışmadıklarına bakınız.

Eğer Pvm konsolunda takılı kalırsanız, el ile Pvm servisini öldürmeyi deneyiniz. Eğer olmuyorsa yetkili kullanıcıya durumu bildirin. Eğer ağdaki bazı makinalar sanal makinanıza eklenmiyorsa , eklenmeyen makinaya bağlanın ve "/tmp" dizinin icindeki sizin kullanıcınıza ait olan bütün pvm ile başlayan dosyaları siliniz. Makinayı bu şekilde eklemeyi deneyiniz. Eğer Pvm programının derlenmesi sırasında zorlanıyorsanız, örnekler dizini içerisindeki ($PVM_ROOT/examples) Makefile.aimk dosyasına bakarak, örnek bir makefile yazabilirsiniz.

Eğer Mpi çalışırken istenilmeyen bir durum oluyorsa, daha ufak boylu dizilerle programınızı test edin. Süreç sayısını azaltarak, kullandığınız çeşiti değişkenlerin değerlerini yazdırın ve sorunun nereden kaynaklandığını anlamaya çalışın.

8 Sonuç

Paralel hesaplama kütüphaneleri, eğer yüksek hesaplama gücüne ihtiyaç duyuyorsanız çok önemlidir. Eğer çok kullanıcı bir ağ ortamında iseniz, bunları güvenli bir şekilde çalıştırıp, bilgisayarların kullanılmadığı zamanlarda hesaplamalarınızı yapabilirsiniz. Eğer probleminizin doğası bölünebilmeye, gerçekten uygunsa ve bölünen parçaların birbileri ile az haberleşmesi gerekiyor ise, ağ ortamında bunları çalıştırmak size çok ucuz maliyetli ve güçlü bir hesaplama ortamı sunabilir. Böylelikle ağınızın uygulama alananı genişletip daha etkin bir kullanıma kavuşabilirsiniz. Bu çeşit kullanım eğitim amaçlı kullanımlar için ucuz maliyeti ve öğrencilere vereceği heyecan nedeniyle düşünülebilir. Eğer daha fazla güç isterseniz, ağ hızınızı arttırıp, daha güçlü makinalar alabilirsiniz. Bu durumda bile maliyet, bir süper bilgisayarınkinden çok daha az olacaktır.

Mpi ve Pvm performansı bu konferanstaki başka bir sunumun konusudur. Hız değerlendirmesi için 49 nolu sunum incelenebilir.

9 Ek Bilgi

Bütün yazılanlar SuSE 8.0 kurulu bir kişisel bilgisayarda denenmiştir. Aynı zamanda ağ ortamı için İTÜ Bilişim Enstitüsü laboratuvarlarını kullanılmıştır. Bu makinaların hepsinde SuSE 8.0 kuruludur. Ancak programların kaynak kodlu kurulumu anlatıldığı için diğer Linux dağıtımlarında da çalışmasını olasıdır.

10 Teşekkür

Bu belgeyi hazırlamamda bana yardımcı olan, Pvm çalıştarılmasını anlamamda yardımcı olan arkadaşım Oğuz Akyol'a , benden desteğini esirgemeyen sayın hocam Metin Demiralp'e , bütün Bilişim Enstitüsü çalışanlarına, aileme ve bütün arkadaşlarıma çok teşekkür ederim.

Kaynaklar

1. Linux İşletim Sistemi Üzerine MPICH Programının kurulması ve MPI'ya giriş - Abbas Ayhan Kanmaz - 1. Serbest Yazılım ve Linux Konferansı ODTU
2. http://www.be.itu.edu.tr/kaynak/kaynak/sy/
3. http://www.ee.ualberta.ca/ kees/Howto-PVM___ee_ualberta_ca.html
4. http://www-unix.mcs.anl.gov/mpi/mpich/
5. http://www.csm.ornl.gov/pvm/pvm_home.html
6. Professional Linux Deployment - Wrox Press Ltd -

7. A User's Guide to MPI, Peter S. Pacheco, Dept. Of Math.University of San Francisco
8. ssh, ssh-keygen, ssh-agent, rsh, rhosts kılavuz (man) sayfaları
9. PVM: Parallel Virtual Machine A Users' Guide for Networked Parallel Computing, MIT Press
10. Bilgisayar Tabanlı Görselleştirmede Paralel Uygulamalar - Hüseyin Kaya, Murat Kaplan, Akademik Bilişim
11. Parallel programming : techniques and applications using networked workstations and parallel computers, Prentice Hall
12. Süper Bilgisayarlar,Bilim ve Teknik Ocak 2001
13. Parallel Numerical Algorithms Ders Notları (2001-2002 Yaz)- Doc. Dr Serdar Çelebi - İTÜ Bilişim Enstitüsü
14. Parallel and Distributed Computing Ders Notları (2001-2002 Güz) - Doc. Dr Hasan Dağ - İTÜ Bilişim Enstitüsü