T.C.

SELÇUK ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

BİLİMSEL HESAPLAMA PROBLEMLERİNİN ÇÖZÜMÜNDE PARALEL HESAPLAMA YÖNTEMLERİNİN KULLANILMASI

Serdar KAÇKA

YÜKSEK LİSANS TEZİ

BİLGİSAYAR MÜHENDİSLİĞİ
ANABİLİM DALI

Şubat-2011
KONYA
Her Hakkı Saklıdır

TEZ KABUL VE ONAYI

Serdar KAÇKA tarafından hazırlanan "Bilimsel Hesaplama Problemlerinin Çözümünde Paralel Hesaplama Yöntemlerinin Kullanılması" adlı tez çalışması 21/02/2011 tarihinde aşağıdaki jüri tarafından oy birliği ile Selçuk Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda YÜKSEK LİSANS TEZİ olarak kabul edilmiştir.

Jüri Üyeleri	**İmza**
Başkan Prof.Dr. Şirzat KAHRAMANLI	.....
Danışman Doç.Dr. Galip OTURANÇ	.....
Üye Doç.Dr. Aşır GENÇ	.....

Yukarıdaki sonucu onaylarım.

Prof.Dr. Bayram SADE
FBE Müdürü

TEZ BİLDİRİMİ

Bu tezdeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edildiğini ve tez yazım kurallarına uygun olarak hazırlanan bu çalışmada bana ait olmayan her türlü ifade ve bilginin kaynağına eksiksiz atıf yapıldığını bildiririm.

DECLARATION PAGE

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Serdar KAÇKA

Tarih: 24.01.2011

ÖZET

YÜKSEK LİSANS

BİLİMSEL HESAPLAMA PROBLEMLERİNİN ÇÖZÜMÜNDE PARALEL HESAPLAMA YÖNTEMLERİNİN KULLANILMASI

Serdar KAÇKA

Selçuk Üniversitesi Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Doç.Dr. Galip OTURANÇ

2011, 73 Sayfa

Jüri
Doç.Dr. Galip OTURANÇ
Prof.Dr. Şirzat KAHRAMANLI
Doç.Dr. Aşır GENÇ

Simülasyon ve onun en önemli bileşeni olan bilimsel hesaplama gün geçtikçe bilim adamlarının ve mühendislerin işini kolaylaştırmaya devam ediyor. Bunu yaparken daha fazla işlem gücü, daha performanslı hesaplama kabiliyeti ihtiyacı artıyor. Bu tezde simülasyonun günümüzdeki kullanımı, bilimsel hesaplama ile olan ilişkisi, bilgisayar performansının sınırları ve bu sınırları aşmak için paralel bilgisayar sistemlerinin önemi ve çeşitleri, matematiksel problemlerin paralel algoritmalar ile nasıl çözülebileceği ve bu çözümlerin bir yazılım sistemi ile nasıl gerçekleştirilebileceği sunulmuştur. Bunun yanında bir paralel bilgisayarın nasıl kurulacağı çalışılmıştır.

Anahtar Kelimeler: Paralel hesaplama, paralel algoritma tasarımı,MPI

ABSTRACT

MS THESIS

USING PARALLEL COMPUTING METHODS FOR SOLVING SCIENTIFIC COMPUTING PROBLEMS

Serdar KAÇKA

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCE OF SELÇUK UNIVERSITY
THE DEGREE OF MASTER OF SCIENCE
IN COMPUTER ENGINEERING

Advisor: Assoc.Prof.Dr. Galip OTURANÇ

2011, 73 Pages

Jury
Assoc.Prof.Dr. Galip OTURANÇ
Prof.Dr. Şirzat KAHRAMANLI
Assoc.Prof.Dr. Aşır GENÇ

Simulation and its main component, scientific computing, has been providing great opportunities to scientists and engineers. While doing this the need for more computing power and computing capabilities is growing. In this thesis, the use of simulation, its relationship with scientific computing, the limits of computer performance and the ways to overcome these limits, solving mathematical problems using parallel computing methods and the ways of implementing these methods has been presented. In addition the ways of building parallel computers has been studied.

Keywords: Parallel computing, parallel algorithm design, MPI

ÖNSÖZ

Yüksek lisans eğitimim ve bu tezin yazım süresince seçtiğim konu ile ilgili önerileriyle yol gösteren ve desteğini esirgemeyen danışmanım sayın değerli hocam Doç. Dr. Galip OTURANÇ'a teşekkür ederim.

Bana gösterdiği samimiyet ve desteklerinden dolayı bilgisayar mühendisliği bölüm başkanı değerli hocam sayın Prof. Dr. Şirzat KAHRAMANLI hocama teşekkür ederim.

Bürokratik işlerle yardım ettiği için ve çalıştığım alanda bana kaynaklar sağladığı için değerli hocam Yard. Doç. Dr. Yıldıray KESKİN'e teşekkür ederim.

Serdar KAÇKA
KONYA-2011

İÇİNDEKİLER

ÖZET ..... **iv**

ABSTRACT ..... **v**

ÖNSÖZ ..... **vi**

İÇİNDEKİLER ..... **vii**

SİMGELER VE KISALTMALAR ..... **ix**

1. GİRİŞ **1**

1.1. Simülasyon 1
1.2. Simülasyon ve Bilimsel Hesaplama 2

2. PARALEL HESAPLAMA **4**

2.1. Giriş 4
2.2. Paralel Hesaplama Donanımları ve Yazılımları 6
2.2.1. Flynn Taksonomisi 6
2.2.2. Paralel bilgisayarlarda iletişim 11
2.2.2.1. Paylaşımlı adres uzayına sahip platformlar 11
2.2.2.2. Mesaj geçiren platformlar 13
2.3. İletişimin Performansa Etkisi 13
2.4. Matematiksel Paralellik 15
2.5. Amdahl Kanunu 17
2.6. Paralel Program Tasarımı 18
2.3.1. Alan ayrıştırması 19
2.3.2. Fonksiyon ayrıştırması 19

3. MESAJ GEÇİRME ARABİRİMİ **20**

3.1. Giriş 20
3.1.1. MPI standartı 22
3.1.2. MPI kullanmanın sebepleri 23
3.2. MPI Yordam Çeşitleri 23
3.2.1. MPI program yapısı 23
3.2.2. MPI veri tipleri 24
3.2.3. Noktadan noktaya iletişim 25
3.2.3.1. Kaynak ve hedef 25
3.2.3.2. Mesajlar 26
3.2.3.3. Mesajların gönderilmesi ve alınması 27
3.2.3.4. İletişim modları ve tamamlanma kriterleri 27
3.2.3.5. Tıkanmalı ve tıkanmasız iletişim 27
3.2.3.5. Tıkanmalı gönderme ve alma 28
3.2.3.6. Tıkanmalı gönderme ve alma örnek uygulaması 29
3.2.3.7. Kilitlenme 30
3.2.3.8. Tıkanmasız gönderme ve alma 30

3.2.3.9. Tıkanmasız alma örnek uygulaması....33
3.2.3.10. Gönderme ve alma modları....34
3.2.4. Toplu iletişim....35
3.2.4.1. Yayımlama işlemleri....36
3.2.4.2. İndirgeme işlemleri....37
3.2.4.3. Bir araya getirme işlemleri....39
3.2.4.4. Saçılma işlemleri....42

5. UYGULAMALAR....45

5.2. Nümerik İntegral....45
5.1.1. Yamuk yöntemi ile nümerik integral....46
5.1.2. Simpson yöntemi ile nümerik integral....47
5.1.3. Paralel nümerik integral....49
5.2. Matris Vektör Çarpımı....51

4. PARALEL BİLGİSAYAR KURULUMU....52

4.1. Paralel Bilgisayar Çeşidi....52
4.2. Kurulumda Kullanılan Bilgisayarlar....53
4.3. İşletim Sistemi....53
4.3.1. Linux işletim sistemi....53
4.3.1.1. Linux işletim sisteminin özellikleri....54
4.3.1.2. Linux dağıtımları....54
4.3.1.3. Fedora dağıtımı....55
4.3.2. Bir Beowulf Kümesinde Linux kullanmanın nedenleri....55
4.3.3. İşletim sistemini kurarken dikkat edilen hususlar....57
4.3.4. Ağ ayarı....57
4.3.4. NTP ayarı....59
4.3.6. SSH ayarı....60
4.4. MPICH2 Kurulumu....62

5. SONUÇLAR VE ÖNERİLER....65

KAYNAKLAR....66

EKLER....67

EK-1 Örnek Uygulamalar Kaynak Kodları....67

ÖZGEÇMİŞ....73

SİMGELER VE KISALTMALAR

API	: Uygulama Programlama Arabirimi (Application Programming Interface)
CPU	: Merkezi İşlem Birimi (Central Processing Unit)
DHCP	: Dynamic Host Configuration Protocol
FIFO	: İlk Giren İlk Çıkar Organizasyonu (First In First Out)
FLOPS	: Saniyedeki Kayan Nokta İşlem Sayısı (Floating Point Operations Per Second)
FTP	: Dosya Aktarma Protokolü (File Transfer Protocol)
GB	: 2^{30} Bayt
GHz	: 10^9 Hz
HPC	: Yüksek Performanslı Hesaplama (High Performance Computing)
IDE	: Bütünleşik Geliştirme Ortamı (Integrated Development Environment)
IEEE	: Elektrik Elektonik Mühendisleri Enstitüsü (Institute of Electrical and Electronics Engineers)
IP	: İnternet Protokolü (Internet Protocol)
IPS	: Saniyedeki Komut Sayısı (Instructions Per Second)
IPX	: Internet Paket Değişimi (Internet Packet Exchange)
ISO	: Uluslararası Standartlar Teşkilatı (International Organization for Standardization)
LAN	: Yerel Alan Ağı (Local Area Network)
MB	: 2^{20} Bayt
MHz	: 10^6 Hz
MIMD	: Çok Komut Çok Veri (Multiple Instruction Multiple Data)
MISD	: Çok Komut Tek Veri (Multiple Instruction Single Data)
MPI	: Mesaj Geçirme Arabirimi (Message Passing Interface)
NFS	: Ağ Dosya Sistemi (Network File System)
NTP	: Ağ Zaman Protokolü (Network Time Protocol)
NUMA	: Tekdüze Olmayan Bellek Erişimli (Non-uniform Memory Access)
PFLOPS	: 10^{15} FLOPS
PVM	: Paralel Sanak Makina (Parallel Virtual Machine)
RAM	: Rasgele Erişimli Bellek (Random Access Memory)
SIMD	: Tek Komut Çok Veri (Single Instruction Multiple Data)
SISD	: Tek Komut Tek Veri (Single Instruction Single Data)
SPMD	: Tek Program Çoklu Veri (Single Program Multiple Data)
SSH	: Secure Shell
TCP/IP	: Transmission Control Protocol / Internet Protocol
TFLOPS	: 10^{12} FLOPS
UMA	: Tekdüze Bellek Erişimli (Uniform Memory Access)
X.25	: Paket değişimli geniş alan ağlarındaki iletişimde kullanılan protocol

1. GİRİŞ

1.1. Simülasyon

Türk Dil Kurumu'nun resmi İnternet sitesinde hizmet veren Güncel Türkçe Sözlüğünde 'simülasyon' kelimesi arandığı zaman sonuç ekranında sırasıyla 'benzetim' ve 'öğrence' kelimeleri listeleniyor. Her iki sonuç da günümüzde simülasyon sistemlerinin kullanılma alanlarını kapsar nitelliktedir.

'Öğrence' olarak simülasyon tıp eğitiminden askeri eğitimine kadar pek çok eğitim alanında kullanılmaktadır. Stanford Üniversitesi'nden Dylan Marks 2005 yılında yayınladığı "Eğitimde Simülasyon Kullanımına Dair Yorumlar" başlıklı yazısında simülasyonun gerçek dünya nesnelerinin yerine geçtiğini söylüyor ve tıp eğitiminde kullanımına dair şu cümleleri sarfediyor: "Simüle edilmiş modelleri tıp okullarında kullanmanın genel faydası bu modellerin gerçek insanların yerine geçebilmeleri ve bu insanları tecrübesiz ellerden uzak tutmamızı sağlamalarıdır. Bu sayede tıp öğrencileri belli bir süreci veya genel bir tanıyı yüzlerce defa pratik yapma imkanına sahip olabilmektedirler (Marks, 2005)."

'Benzetim' olarak simülasyon doğal süreçlerin bir bilgisayar sisteminde modellenip belli sonuçların elde edilmesi ve bu sonuçlara göre bazı bilimsel çıkarımların yapılması gibi bilimsel faaliyetlerde kullanım alanı bulmaktadır. Ayrıca tasarım ve üretim süreçlerinde de bu sistemlerin sağladığı faydalar gün geçtikçe artmaktadır. Scientific Computing World dergisininin Aralık 2006/Ocak 2007 sayısında David Robson, "Simülasyon Zaman Kazandırır" başlıklı makalesinde yazılım uygulamalarının mühendislik ve fizik alanlarındaki kullanımını inceliyor. "Astrofizik, otomotiv endüstrisi ve biyotıp gibi farklı alanlardaki araştırmacılar fiziksel simülasyon ve modelleme yazılımlarını ürünlerin tasarımını ve testini kolaylaştırmak için kullanmaktadırlar. Sağlanan faydalar gelişmiş tasarımlar, düşük üretim maliyetleri ve daha fazla yarar gibi olarak sıralanabilir: Viasys Healthcare firmasına göre, küçük çocuklar için kullanılan bir vantilatör cihazını yeniden tasarlamak sürecinde kullandıkları CD-adapco firmasının hesaplamalı akışkanlar dinamiği simülatörü Star-CD yazılımı, tasarım aşamasını bir yıldan üç güne düşürmüş ve firmayı 250.000 dolarlık bir masraftan kurtarmış." Gazların ve sıvıların akışını yöneten Naiver-Stokes denklemlerinin bazı durumlarda çözümlerini bulmak imkansız olmaktadır. Bu gibi

durumlarda bir bilgisayarın sağladığı nümerik çözümler tek olası yol olarak karşımıza çıkmaktadır (Robson, 2007).

1.2. Simülasyon ve Bilimsel Hesaplama

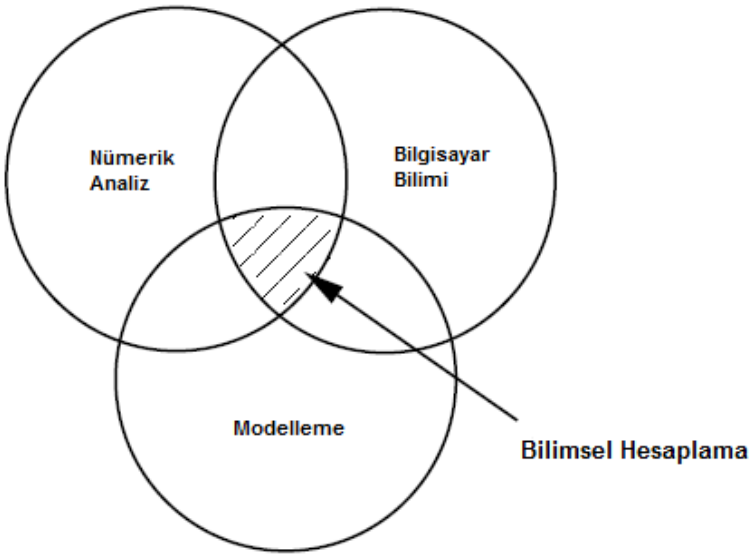
Bilim ve mühendislikte yapılan araştırmalar ve kullanılan teknolojiler bakımından büyük bir dönüşüm olagelmektedir. Günümüzün bilim insanı ve mühendisi artık zamanının büyük bir kısmını bir bilgisayar ekranının karşısında harcamaktadır. Eskiden laboratuar ve çalıştaylarda yapılan çalışmalar günümüzde çok hızlı bilgisayarlarda çalıştırılan simülasyon yazılımları ile yapılmaktadır (Karniadakis ve ark., 2003). Bilişim teknolojisinin üstel bir davranışta gelişmesi sonucu bilimsel yaklaşımdaki bir paradigma değişiminden bahsedebiliriz. Gözlemlerin ve deneylerin hakim olduğu klasik bilimsel yaklaşımdan simülasyon yaklaşımına bir geçiş sözkonusudur.

Simülasyon yaklaşımında ilk adım fiziksel sistemin doğru bir temsilini seçmektir. Önemli denklemleri ve ilgili sınır şartlarını ortaya çıkarmak için varsayımlar tutarlı olmalıdır. Korunum yasaları sağlanmalıdır; entropi şartı ihlal edilmemelidir. İkinci adım olarak sürekli süreci kesikli hale getirmek için doğru bir algoritmik süreç geliştirilmelidir. Üçüncü adımda bu hesap verimli bir şekilde yapılmalıdır. Hesaplamanın ne kadar verimli olduğu problemin ne kadar gerçekçi bir çözüme kavuştuğuna ve dolayısıyla sonuçların uygulamalar için ne kadar kullanışlı olduğuna bağlıdır. Dördüncü adımda fiziksel deneyler aracılığıyla doğrudan bir doğrulama imkanı olmayan durumlar için sonuçların tutarlılığını değerlendirmektir. Bu gibi durumlara örnek olarak astrofizik ve nanoteknoloji bilim alanları verilebilir. Son olarak düzgün bir bilgisayar grafiği yöntemi kullanarak simüle edilmiş süreci görselleştirmek simülasyon döngüsünü tamamlar (Karniadakis ve ark., 2003).

Simülasyon biliminin en önemli bileşeni bilimsel hesaplama alanıdır. Aslında önceleri nümerik analiz olarak adlandırılan çalışma alanı için bugün bilimsel hesaplama adı kullanılmaktadır. Bu alan, hesaplamalı bilimlerde ve mühendislikte ortaya çıkan matematiksel problemleri çözmek için kullanılan algoritmaların tasarımı ve analizi ile ilgilenir (Heath, 1997). Bilgisayar biliminden ve modellemeden bir katkı söz konusudur. Bir başka tanımı da şöyle ifade edileblir: “bilim ve mühendislikteki problemleri adresleyen matematiksel modelleri bir veya birden fazla bilgisayarda çözmek için gereken araçlar, teknikler ve teorilerin toplamına bilimsel hesaplama denir (Golub ve

ark., 1992).” Bilimsel hesaplama doğaları gereği sürekli olan değişkenlerin (zaman, mesafe, hız, sıcaklık, basınç gibi) oluşturduğu denklem ve fonksiyonlar ile ilgilenir.

Şekil 1.1.'de bilimsel hesaplama üç alanın ortak bir çalışması olarak gösterilmiştir (Karniadakis ve ark., 2003). Buna göre, bilimsel hesaplama ancak ve ancak disiplinler arası bir çalışma sonucu yapılabilecek bir çalışmadır denebilir.



Şekil 1.1. Bilimsel hesaplamanın üç alanın ortak bir çalışması olarak gösterilmesi

Bu alanda yapılan çalışmaları özetlemek gerekirse, sürekli matematiğin problemlerini bir sonuca yaklaşarak tekrarlı bir süreç ile çözmektir diyebiliriz. Bu sonuç olabildiğince hassas ve tutarlı olmalıdır.

2. PARALEL HESAPLAMA

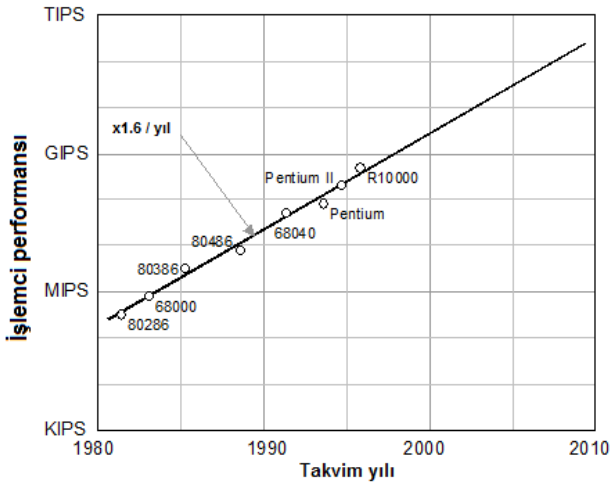
2.1. Giriş

Bilimsel hesaplama ve mühendislik alanlarında ortaya çıkan problemlerin karmaşıklıkları ve veri gereksinimleri devamlı olarak artmaktadır. Bundan dolayı performansı daha yüksek olan bilgisayar arayışı hala devam etmektedir. Tarihsel olarak bilgisayarların hesaplama güçleri bu artışla başa çıkamayıp bundan dolayı paralel hesaplama yöntemleri geliştirilmiştir.

Paralel hesaplamanın ardında birden çok motivasyon vardır.

1965 yılında Gordon Moore tarafından formüle edilen ve **Moore Yasası** olarak literatüre geçen yasaya göre devre karmaşıklığı her 18 ay iki kat artar. Bu aynı artışın işlemci performansı için de geçerli olduğu anlamına gelmektedir. İşlemci performansını ister IPS (Saniyedeki Komut Sayısı – *Instructions Per Second*) ile ölçelim, ister FLOPS (Saniyede Gerçekleştirilen Kayan Nokta Sayılı İşlem Sayısı – *Floating-Point Operations Per Second*) ya da işlemcilerin performansını gerçek uygulamalar üzerinde ölçmeye çalışan güçlü benchmark yazılımlarını kullanalım Moore Yasasının geçerli olduğunu göreceğiz. Şekil 2.1.'de 1980 yılından beri üretilen işlemcilerin performansının Moore Yasasına uyduğu gösterilmektedir (Parhami, 2002).

Moore Yasasının yakın gelecekte geçerli olacağı beklense de erişilecek bir sınır vardır. Bu sınır doğa yasalarının dayattığı bir sınırdır: bir tel üzerindeki sinyal yayılma hızı sonludur. Buna **ışık hızı argümanı** denir.



Şekil 2.1. İşlemci performansının Moore Yasasına uyan artış grafiği

Bu sınıra ulaşıldığı zaman performans artırmanın tek yolu birden çok işlemci kullanmaktır. Aynı argüman paralel işlemcilerin birbirleriyle haberleşirken bir sınıra ulaşılacağına dair bir iddia için de kullanılabilir. Ancak, her düşük seviye işlem için böyle bir haberleşmeye ihtiyaç duyulmayacağından dolayı, burdaki sınırın ciddiyeti çok düşüktür. Gerçekte, çoğu uygulamada, ard arda gelen iki iletişim adımı arasında büyük sayıda hesaplama adımı gerçekleştirilebilir; böylece iletişim masrafı amortize edilmiş olur.

TFLOPS (10^{12} FLOPS) veya PLOPS (10^{15} FLOPS) performansına sahip süper bilgisayarlara hangi uygulamalarda ihtiyaç duyulmaktadır? Ordudaki, uzay araştırmalarındaki ve iklim modellemedike uygulamalar bilinen uygulamalardır. Bunların yanında araç kaza simülasyon uygulamaları, ilaç tasarım uygulamaları, tümleşik devre tasarım uygulamaları, bilimsel görselleştirme ve çoklu medya uygulamaları örnek olarak verilebilir. Varolan süperbilgisayarların performansının yetmediği bir uçağın aerodinamik simülasyonunun yapan uygulamalar, küresel klimayı onlarca yıl modelleyen uygulamalar ve gelişmiş maddelerin atomik yapılarını inceleyen uygulamalar örnek olarak sayılabilir (Parhami, 2002).

Genel olarak bir uygulama şu üç sebepten dolayı birden daha falza hesaplama gücüne ihtiyaç duyar (Gropp ve ark., 2003):

- Gerçek zamanlı kısıtlar: Buna örnek olarak hava tahmini programları verilebilir. Pazartesi günü ile ilgili tahmini salıya yetiştirmenin herhangi bir pratik faydası yoktur. Bir diğer örnek olarak bir deneyde üretilen verilerin en azından üretildiği hızda işlenmesi (veya saklanması) gerekmektedir ki daha sonraki analizlerde bu verilerin ve dolayısıyla deneyin bir anlamı olmuş olsun.
- Verimlilik: Verimlilik (*throughput*) belirli bir zaman periodunda bir bilgisayarın yapabileceği iş miktarıdır. Bazı simülasyonların hesaplama gücüne o kadar çok ihtiyacı vardır ki hesaplamayı bir bilgisayar ile bitirmeye çalışmak günler hatta aylar sürebilir.
- Bellek: Bazı simülasyonların bir bilgisayarın sınırlarını kat be kat aşan bellek ihtiyaçları vardır.

Daha büyük performanslı donanımlar kullanarak, problemleri çözen algoritmaları optimize ederek ve/veya paralel hesaplama yöntemleri kullanmaya imkan sağlayan paralel bilgisayarlar kurarak daha fazla işlem gücü elde edilebilir.

Paralel hesaplama bir bilgisayarın kabiliyetlerini aşan karmaşık bir problemi çözmek için birden çok hesaplama kaynağının aynı anda kullanılmasını kapsayan bir hesaplama biçimidir. Hesaplama kaynakları,

- birden çok işlemciye sahip bir bilgisayarı,
- birbirleri ile bağlı birden çok bilgisayarları (paralel bilgisayar) veya
- her ikisinin kombinasyonu

içerebilir. Problem eş zamanlı olarak çözülebilecek daha küçük parçalara ayrılır. Yani problemin çözümünü belirleyen algoritmanın paralelleştirilmesi gerekmektedir. Daha sonra bu paralelleştirilen kısımlar eş zamanlı süreçler tarafından işlenip problemin genel çözümüne katkı sağlanır.

Paralel hesaplama paradigmalarında süreç kavramı önemlidir. Süreç bir fiziksel işlemci üzerinde çalışan özerk bir program veya altporgramdır. Bir sürecin eriştiği yerel bir depolama alanı vardır. Bir program çalışma zamanının herhangi bir anında birden çok süreci kapsar hale geliyorsa sözkonusu program paralel bir programdır (Pacheco, 1997).

Verimli bir paralel hesaplama için hem donanım hem de yazılım kabiliyetlerine ihtiyaç vardır. Süreçleri belirleyen, oluşturan ve yok eden yollar olmalıdır. Süreçler arasındaki iletişimi belirleyen protokoller sağlanmalıdır. Süreçler arasındaki karşılıklı bağlantılar bu süreçler arasındaki iletişimi hızlı bir şekilde yerine getirmelidir.

2.2. Paralel Hesaplama Donanımları ve Yazılımları

Paralel hesaplamayı gerçekleştirmek için çok çeşit donanım mimarisi ve bu mimarileri destekeyen yazılım modelleri mevcuttur. Donanım mimarilerini mantıksal bir sıraya sokmak zor olabilir. Ama yine de bir yerden başlamak gerekirse bilgisayar mimarilerini sınıflamak için Michael J. Flynn tarafından 1966 yılında ortaya konan ve Flynn Taksonomisi olarak bilinen sınıflamadan başlanabilir.

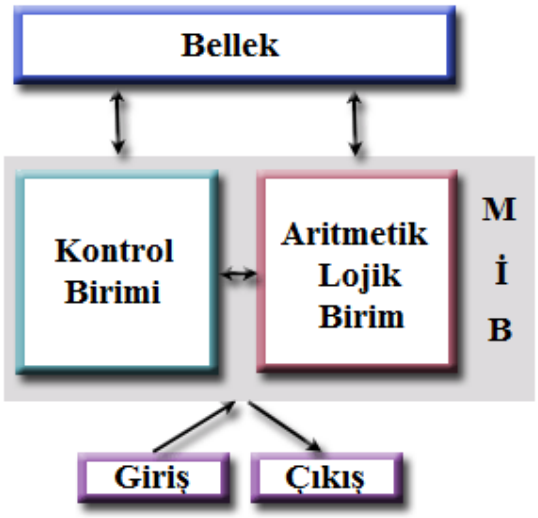
2.2.1. Flynn Taksonomisi

Bu taksonomide bilgisayar mimarileri **komut** ve **veri** olmak üzere iki bağımsız boyut üzerinden sınıflanırlar. Bu boyutlardan her biri **tek** veya **çok** olmak üzere iki olası durumda olabilir. Dolayısıyla ortaya dört durum daha doğrusu dört sınıf çıkmaktadır. Çizelge 2.1.'de bu sınıflar gösterilmektedirler.

Çizelge 2.1. Flynn Taksonomisi

	Tek Komut	Çok Komut
Tek Veri	SISD	MISD
Çok Veri	SIMD	MIMD

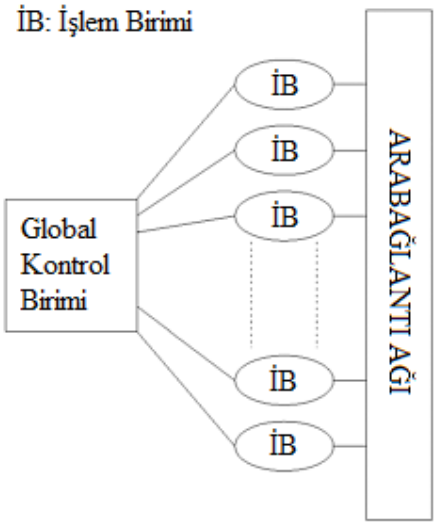
Bu taksonomideki en basit mimari SISD (*Single-Instruction Single-Data*) mimarisidir. Buna örnek olarak klasik von Neumann makinası (Şekil 2.2.) gösterilebilir.



Şekil 2.2. MİB, Bellek ve Giriş/Çıkış birimlerinden oluşan von Neumann mimarisi, SISD mimarisinin bir örneğidir

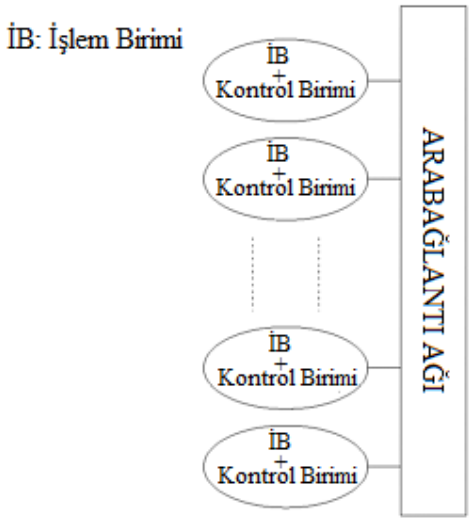
MISD (*Multiple-Instruction Single-Data*) sistemleri genellikle hata toleranslı uygulamalarda kullanılırlar. Bu sistemlerde heterojen sistemler aynı veri üzerinde çalışırlar. Buna örnek olarak uzay mekiği uçuş kontrolü bilgisayarı verilebilir.

SIMD (*Single-Instruction Multiple-Data*) sistemlerde ise tüm işlem birimleri (*porcessing unit*) herhangi bir saat döngüsünde senkron bir şekilde aynı komutu işlerler. Bütün işlem birimleri bir kontrol birimi tarafından kontrol edilirler. Şekil 2.3.'de bu sistem gösterilmektedir (Grama ve ark., 2003). Her işlem birimi farklı veri elemanını işleyebilir. Bu makinaların iki çeşidi vardır: işlemci dizileri (Connection Machine CM-2, ILLIAC IV) ve vektör iş hatları (IBM 9000, Cray X-MP, ETA 100). Grafiksel işlemci birimi (GPU) olan çoğu bilgisayar bu mimariyi içinde barındırır (Barney, 2010).



Şekil 2.3. SIMD mimarisi

Flynn Taksonomisinin karşı ucunda MIMD (*Multiple-Instruction Multiple-Data*) mimarisi bulunur ve kendi verileri ile işlem yapan özerk işlemcilerden oluşur. Dağıtık sistemler genelde bu sınıfa girerler.

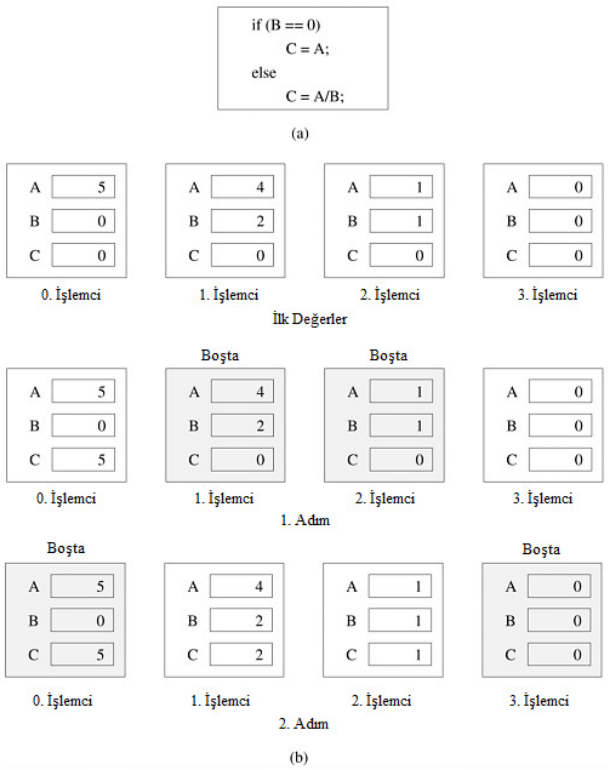


Şekil 2.4. MIMD mimarisi

Şekil 2.4.'de gösterilen bu sistemde SIMD mimariden farklı olarak her işlem birimi diğerlerinden bağımsız olarak farklı bir program çalıştırabilir (Grama ve ark., 2003). SIMD sistemleri senkron çalışırken MIMD sistemleri asenkron bir yapıya sahiptirler.

MIMD mimarisinin özel bir çeşidi SPMD (Tek Program Çok Veri – *Single-Program Multiple-Data*) modelidir. Bu model aslında MIMD sistemlerini programlama yaklaşımlarından (Pacheco, 1997) birini temsil eder ve farklı veriler üzerinde çalışan aynı program örneklerine dayanır (Grama ve ark., 2003). Daha sonra kurulumu ayrıntılı açıklanacak Beowulf Kümeleri de genellikle bu modele göre programlanırlar.

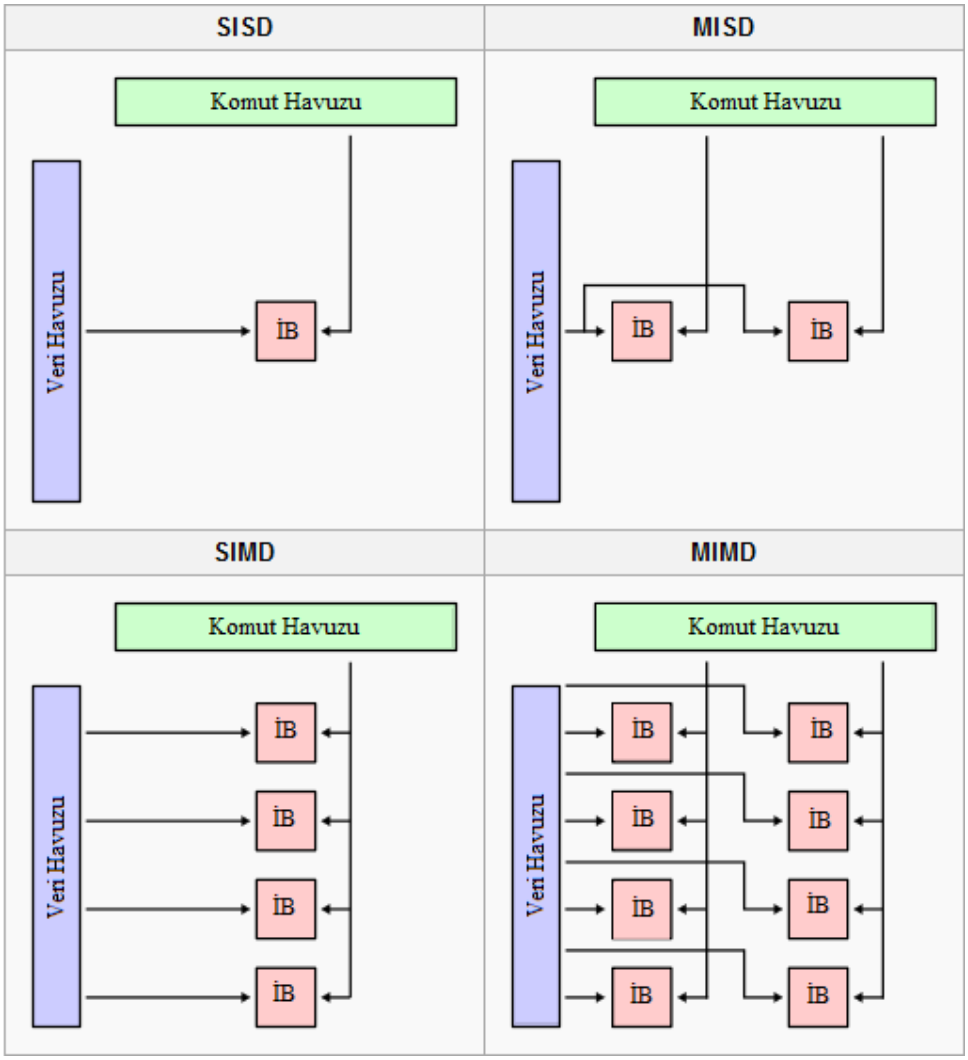
SIMD bilgisayarların tek kontrol birimi olduğu için MIMD bilgisayarlardan daha az donanım bulundururlar. Ayrıca SIMD bilgisayarlarda tek program kopyası depolandığı için daha az bellek gerektirirler. Daha az donanım bulundurması daha basit bir mimariye sahip olduğu anlamına gelmez. Aslında özel donanım mimarilerine sahip olmaları, ekonomik faktörler, tasarım kısıtları ve uygulama özellikleri gibi etkenlerden dolayı SIMD bilgisayarlar çok popüler değillerdir. Diğer taraftan SPMD paradigmasını destekleyen platformlar kullanıma hazır ucuz bileşenlerden kısa zamanlar içinde kurulabilirler. SIMD bilgisayarlar üzerinde çalışan programlarda çok sayıda **if-else** yapısı varsa bu mimarilerde kaynaklar verimli kullanılmaz (Grama ve ark., 2003).



Şekil 2.5. SIMD bir makinada şartlı ifadenin çalıştırılması: (a) şartlı ifade; (b) ifadenin iki adımda icrası

Şartlı ifadelerin olmasının kaynak kullanımına nasıl etkilediği Şekil 2.5.'te gösterilmiştir. (a) alt şeklindeki şartlı ifade iki adımda gerçekleşir. Geçekleşme sırası (b) alt şeklinde gösterilmiştir (Grama ve ark., 2003).

Son olarak Şekil 2.6.'da bu mimariler işlem birimleri ve veri ile komut boyutları açısından resmedilmişlerdir. Burada "İB" olarak gösterilen şekiller işlem birimlerini temsil etmektedirler. Sadece SISD sisteminde tek bir işlem birimi mevcuttur.



Şekil 2.6. Flynn Taksonomisinin işlem birimleri ve veri havuzu/komut havuzu açısından gösterimi

2.2.2. Paralel bilgisayarlarda iletişim

Paralel süreçlerin birbirleriyle iletişim kurup veri alışverişinde bulunmaları gerekir. Bunun iki temel biçimi vardır:

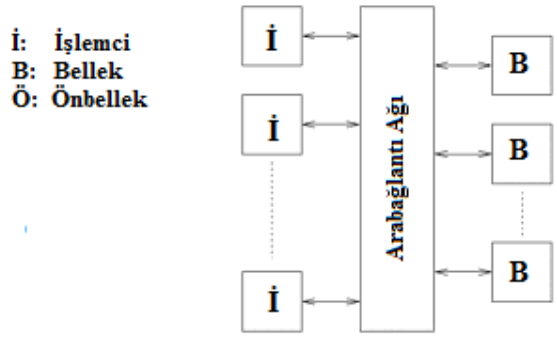
- Ortak bir veri uzayına erişmek.
- Mesajlarla haberleşmek.

2.2.2.1. Paylaşımlı adres uzayına sahip platformlar

Paylaşımlı adres uzayına (*shared-address-space*) sahip paralel bilgisayarlarda tüm işlemciler ortak bir veri uzayına erişirler. İşlemciler bu veri alanındaki objeleri güncelleyerek birbirleri ile haberleşirler. Paylaşımlı adres uzayına sahip platformlardan bellek her işlemciye özel olabilir (yerel) veya tüm işlemciler ortak bir belleğe erişirler (global). İşlemcilerin bellekteki bir kelimeye erişim sürelerine göre bu sistemler ikiye ayrılırlar:

- UMA (Tekdüze Bellek Erişimli – *Uniform Memory Access*) Sistemler
- NUMA (Tekdüze Olmayan Bellek Erişimli – *Non-uniform Memory Access*) Sistemler

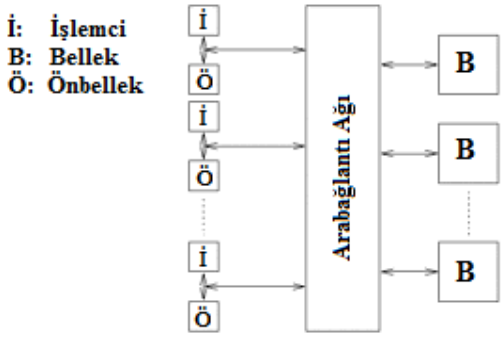
UMA sistemlerde işlemcilerin bellekteki bir kelimeye erişme süreleri aynıdır. Şekil 2.7.'de ve Şekil 2.8.'de iki UMA sistemi gösterilmiştir. Bunlardan ikincisinin işlemcilerinde önbellek de bulunur.



Şekil 2.7. UMA bilgisayar

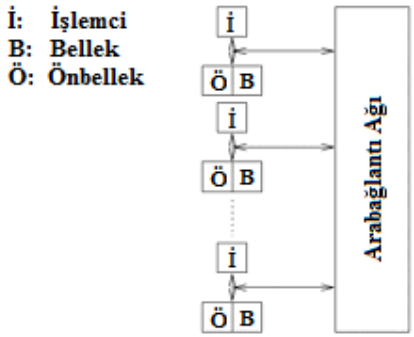
Önbellek bulunması burada herhahngi bir işlemcinin diğerlerine göre belleğe daha hızlı erişeceği ve UMA tanımına ters düştüğü anlamına gelmez. UMA ve NUMA

tanımları önbelleğe erişime göre değil bellek erişimine göre belirlenmişlerdir ve Şekil 2.8.'de gösterilen sistemde tüm işlecilerin önbellek hiyerarşisi vardır (Grama ve ark., 2003).



Şekil 2.8. Önişlemcili UMA bilgisayar

NUMA sistemlerde ise her işlemcinin bellekteki bir kelimeye erişim süresi farklılık arz eder. Şekil 2.9.'da önbellekli bir NUMA bilgisayarı şematik olarak gösterilmiştir. SGI Origin 2000 ve Sun Ultra HPC sunucuları NUMA sistemlere birer örnek olarak gösterilebilir (Grama ve ark., 2003).



Şekil 2.9. Önişlemcili NUMA bilgisayar

UMA ve NUMA platformları arasındaki ayrım önemlidir. Eğer, lokal belleğe erişmek global belleğe erişmekten daha ucuz ise algoritmalar ve veri yapıları bu lokalliği dikkate almalıdırlar. Global bellek uzayının olması durumunda bu platformların programlanması daha kolay olur. Programcı seri bir program yazıyormuş gibi programını yazar.

Paylaşımlı adres uzaylı bilgisayar ile paylaşımlı bellekli bilgisayar karıştırılmaması gereken iki farklı kavramdır. Bunlardan ilki daha çok mantıksal bir

organizasyonu çağrıştırırken ikincisi daha çok fiziksel ortamdaki farklılıklara dayanır. Paylaşımlı bellekli bir bilgisayarda bellek fiziksel olarak işlemciler arasında paylaşılır. Her işlemcinin herhangi bir bellek kısmına erişimi eşittir. Bu UMA sistemine denktir. Bunun karşıtı olan dağıtık bellekli bilgisayarlarda farklı bellek kısımları fiziksel olarak farklı işlemcilere ayrılmışlardır. Her iki fiziksel modelde de paylaşımlı veya ayrık adres uzaylı veya paylaşımlı adres uzaylı platformlar gerçekletirilebilir (Grama ve ark., 2003).

2.2.2.2. Mesaj geçiren platformlar

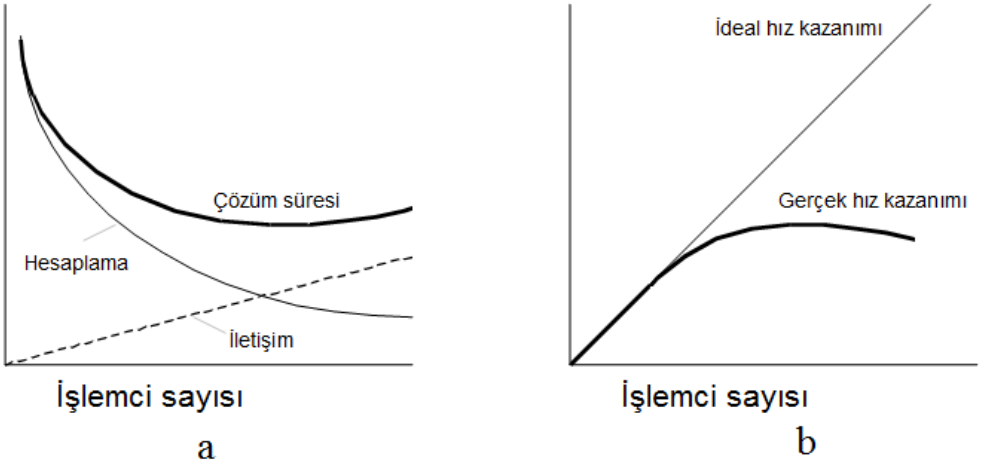
Bu çeşit platformlarda birden çok işleyen birimin (süreç) kendi adres uzayı vardır. İşlemci ve belleği birlikte bir düğümü tanımlar. Farklı düğümlerde çalışan süreçler arasındaki etkileşimler mesaj alış verişleri ile yapılır. Mesaj geçiren ismi burdan gelmektedir. Mesaj alışverişi sonucu verilerin, işlerin aktarımı süreçler arasındaki eylemlerin senkronizasyonu sağlanır. En genel hali ile bu platformlarda her süreçte farklı bir progam çalıştırılabilir. Ancak, SPMD yaklaşımı ile kullanımları daha yaygındır.

Bu programlama paradigmasındaki temel işlemler **gönder** (*send*) ve **al** (*receive*) işlemleridir. Gönderme ve alma işlemlerinde hedef ve kaynak adreslerin bilinmesi gerektiğinden bu platformlarda **ben kimim** (*whoami*) ve programı çalıştırına toplam süreç sayısını veren **süreç sayısı** (*numprocs*) işlemler sağlanmalıdır. Bu dört işlemle mesaj geçiren platformda çalışan herhangi bir paralel program yazılabilir. Farklı mesaj geçirme kütüphaneleri mevcuttur. Bunlar arasında yaygın kullanılanlar MPI (Mesaj Geçirme Arabirimi – *Message Passing Interface*) ve PVM (Paralel Sanal Makinalar – *Parallel Virtual Machines*) kütüphaneleridir (Grama ve ark., 2003).

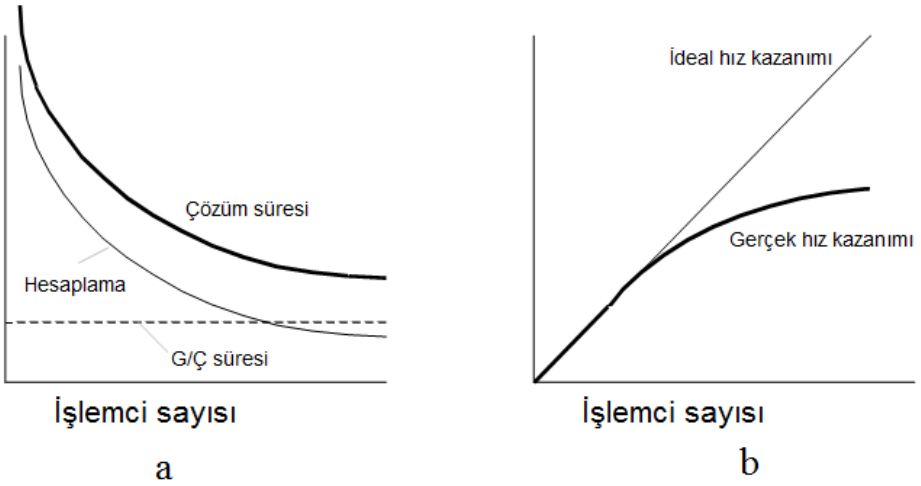
2.3. İletişimin Performansa Etkisi

İşlemci sayısı arttıkça toplam çözüm zamanında her zaman mı bir düşüş söz konusudur? Bu sorunun cevabı hayırdır. MIMD yapılı bir paralel bilgisayarda geliştiren bir çözümdeki toplam süre bilgisayarların birbirleri ile haberleşmeleri için geçen süreye (**iletişim süresi**) ve işlemleri yapmaları için gereken süreye (**hesaplama süresi**) bağlıdır. Aşağıdaki şekilde, belli bir işlemci sayısından sonra işlemci eklemenin toplam çözüm süresini artırdığı gösterilmektedir. Şekil 2.10'da bu durum gösterilmiştir.

Bir de yukarıdaki çözüme G/Ç süresi eklendiği zaman ne olduğuna bakalım. Basitlik sağlaması açısından G/Ç süresinin sabit olduğunu varsayarsak ve iletişim süresini hesaba katmazsak, işlemci sayısı arttıkça hesaplama kısmının hızlanmasına rağmen toplam çözüm süresinde G/Ç süresi gittikçe daha büyük bir dilim tutmaya başlayacaktır. Bu süreç Şekil 2.11.'de gösterilmiştir. Örneğin G/Ç süresi 100 saniye sürüyorsa o zaman hesaplama kısmını 1 saniyede yapmak veya 0.1 saniyede yapmak arasında çok bariz bir fark olmaz. Hesaplamaların bu gibi "sıralı" veya "paralel olmayan" kısımları, paralel hesaplama ile elde edilecek hızlanmayı ciddi şekilde sınırlandırmaktadırlar (Parhami, 2002).



Şekil 2.10. İletişim hızı ile hesaplama hızı arasındaki denge; (a) Çözüm süresi, (b) Hız kazanımı



Şekil 2.11. Sabit bir G/Ç süresinin etkisi; (a) Çözüm süresi, (b) Hız kazanımı

2.4. Matematiksel Paralellik

Yüksek derecede paralelliğe sahip birçok matematiksel işlem vardır. Paralellik dendiği zaman bu işlemlerin aynı anda ve birbirinden bağımsız yapılabilmeleri anlaşılmalıdır. Örnek olarak x ve y iki vektör olmak üzere c vektörünü oluşturacak şekilde bu iki vektörün çarpımına bakalım:

$$c_i = x_i y_i, \quad i = 1, \ldots, N.$$

Burada her çarpım eş zamanlı yapılabilir dolayısıyla bu çarpımdaki her terimin ayrı bir makine tarafından icra edildiğini düşünebiliriz. Burda önemli olan veriler arasında bir bağımlılığın olmaması hususudur. Bu işlem, **mükemmel matematiksel paralelliğe** bir örnektir. *N* adet sıralı ikilinin her birinden en büyüğünü bulurken de, örneğin $max(x_i, y_i)$, $i = 1, 2, \ldots, N$ işleminde de bu paralellik sözkonusudur. Ancak, yukarıdaki sıralı ikililerden mutlak en büyüğünü bulmak istersek o zaman veriler arasında bir bağımlılık kurmuş oluruz. Artık bu problem mükemmel matematiksel paralelliğe sahip değildir. Aynı şey iki vektörün iç çarpımını yaparken de geçerlidir. Bu çarpma işlemi, tüm çarpanların toplamını gerektirir ve bu işlem **seri** bir işlemdir.

Bir başka örnek olarak aşağıdaki polinomun bir x_0 noktasındaki hesaplanmasına bakalım:

$$p(x) = 1 + 2x + 3x^2 + 4x^3 + x^4$$

Basit bir hesap yapmak gerekirse her terimdeki kuvveti teker teker hesaplamak gerekir, çünkü bu yapılmazsa o zaman ek belleğe ihtiyaç duyulur. Ancak bu ifadeye *Horner kuralı* uygulanırsa yukarıdaki polinom şu şekilde yazılabilir:

$$p(x) = 1 + (2 + (3 + (4 + x)x)x)x)$$

Bu son polinom *(N-1)* adet çarpımla ve *N* adet toplamla hesaplanabilir. Bilimsel hesaplamalarda bu tarz optimizasyonlar önemlidir.

Yukarıdaki iç çarpım örneğine bir paralelleştirme uygulamak istersek toplamı iki küçük toplama ayırırız. *N* sayısının çift olduğunu varsayarsak toplam

$$\sum_{i=1}^{N/2} x_i y_i \qquad \sum_{i=N/2+1}^{N} x_i y_i$$

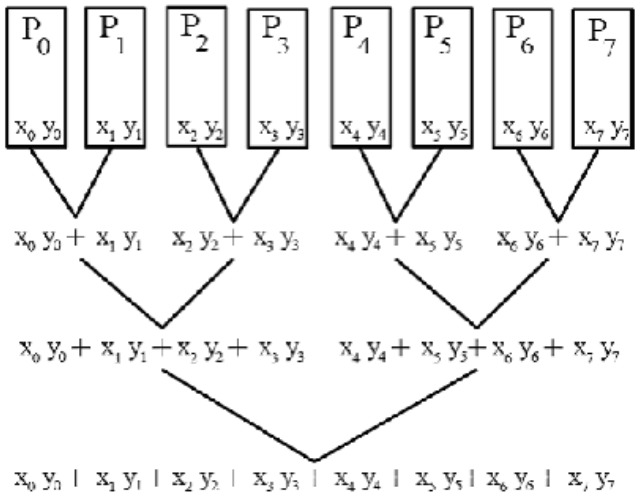
şekline dönüşür. İki toplamı birbirinden bağımsız toplayabiliriz ondan sonra iki sonuca bir toplam daha uygulayıp sonuç toplamı bulmuş oluruz. Bilgisayar bir toplamı yerine getirmek için δt zaman harcarsa ilk yöntemde toplam zaman $T_1=(N-1)\,\delta t$ olur. Problemi iki alt probleme böldükten sonra gereken toplam zaman ise $T_2=(N/2-1)\,\delta t + \delta t + C$ olur. Burada C iki alt toplamda gelen sonuçları getirmek için geçen süredir. Bu yöntemin hız kazanımını (*speed-up*) şu oranla gösteririz:

$$S_2 = \frac{T_1}{T_2} = \frac{N-1}{N/2 + C/\delta t} \tag{2.1}$$

$C/\delta t$ oranı çok küçük olursa o zaman çok büyük N için teorik *maksimum* hıza varılabilir bu da $S_2 \approx 2$ değeridir.

Bu örnek başta seri olarak görülen bir örnekten paralelliğin nasıl çıkartılabileceğini göstermektedir. Bu parçalama işi içe doğru devam edebilir. Bu yaklaşımın adı **böl ve yönet** (*divide and conquer*) yaklaşımıdır ve bu yaklaşım paralel işleme düşüncesinin temelidir. Bu yaklaşım bir avantajı daha vardır: nümerik stabilite. Çok sayıda değeri seri bir şekilde toplarsak hata birikimi önemli bir seviyeye gelebilir. Ama bu algoritmada hata birikimi daha yavaş olur.

Yukarıda iki işlemci için verilen örneği genelleştirmeye çalışalım. P adet işlemci, $P=N$, $N=2^q$ olsun. Son eşitlik, *nokta çarpımını* her işlemcideki iki sayının toplamına indirgemek içindir. $N=8$ olduğunda toplam adım sayısı $q=3$ olur ve bu toplam Şekil 2.12'de gösterilmiştir:



Şekil 2.12. N=8 olduğunda nokta çarpımı

Bu durumda hızlanma katsayısı

$$S_P = \frac{T_1}{T_P} = \frac{(N-1)\delta t}{q\delta t + qC} \tag{2.2}$$

olur. Burada qC iletişim hızıdır. $\alpha = C/\delta t$ göreceli zaman ise, hızlanma faktörünü problem büyüklüğü cinsinden veya işlemci sayısı cinsinden yazabiliriz:

$$S_P = \frac{N-1}{(1+\alpha)\log_2 N} = \frac{P-1}{(1+\alpha)\log_2 P}\,. \tag{2.3}$$

(2.3) denkleminden görüldüğü gibi iletişim sıfır olsa bile

$$S_P = \frac{P-1}{\log_2 P} < P \tag{2.4}$$

olacağından hiçbir zaman işlemci sayısı kadar hızlanma elde edilmez. Paralel verimlilik de aşağıdaki denklem ile tanımlanır:

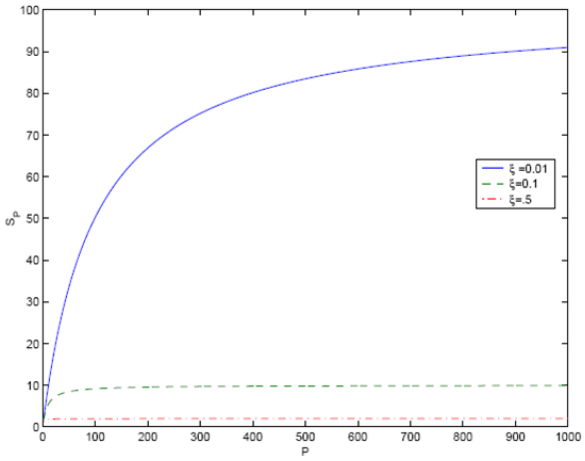
$$\eta_P = \frac{P-1}{P\log_2 P} \tag{2.5}$$

2.5. Amdahl Kanunu

Hızlanma faktörü için daha genel bir model 1967 yılında Gene Amdahl tarafından sunuldu. Bu kanun bir programın ξ kadar bir yüzdesinin paralelleştirilemediğini varsayar, dolayısıyla geri kalan (1 – ξ) miktarı mükemmel paralellik arz eder. Geri kalan tüm iletişim gecikmeleri dikkate alınmazsa bu kanun aşağıdaki denklem ile ifade edilir:

$$S_P = \frac{T_1}{[\xi + (1-\xi)/P]T_1} = \frac{1}{\xi + \frac{1-\xi}{P}} \tag{2.6}$$

Bu kanuna göre P sonuza yaklaştığı zaman S_p de $1/\xi$ değerine yaklaşır, ideal durum asla oluşmaz. Şekil 2.13'te farklı ξ değerleri için S_p grafiği gösterilmiştir.



Şekil 2.13. Farklı ξ değerlerinin için S_p grafiğine etkisi

2.6. Paralel Program Tasarımı

Paralel programları tasarlamak ve geliştirmek genelde manüel bir süreç şeklinde ilerler. Paralelleştirmeyi belirlemek ve gerçekleştirmek genelde programcının sorumluluğundadır. Bu süreç zaman alıcı, karmaşık, hataya yatkın ve tekrarlı bir süreçtir. Bunun yanında zaman içinde programcının işini kolaylaştıran bazı araçlar geliştirilmiştir (Barney, 2010).

Paralel program geliştirirken ihtiyaçlardan biri hesaplama algoritmalarının ve giriş verilerinin eş zamanlı çalışabilecek altprogramlara nasıl ayrıştırılabileceğini belirlemektir. Bu sürece **problem ayrıştırması** (*problem decomposition*) denir.

İki çeşit ayrıştırma sözkonusudur:

- Alan ayrıştırması
- Fonksiyon ayrıştırması

Çoğu program bu iki çeşidin bir karışımından ibarettir (Pacheco, 1997).

2.3.1. Alan ayrıştırması

Alan ayrıştırması (*domain decomposition*) veriler aşağı yukarı aynı parçalara ayrılırlar ve süreçler ile eşleştirilirler. Her süreç sadece kendisine atanmış veri kısmı ile çalışır. Her süreç genelde veri üzerinde aynı komutları çalıştırır. Bu yaklaşım **veri paralelleştirilmesi** (*data parallelism*) olarak da bilinir. Süreçler belli aralıklarda birbirleri ile haberleşip veri alışverişinde bulunurlar.

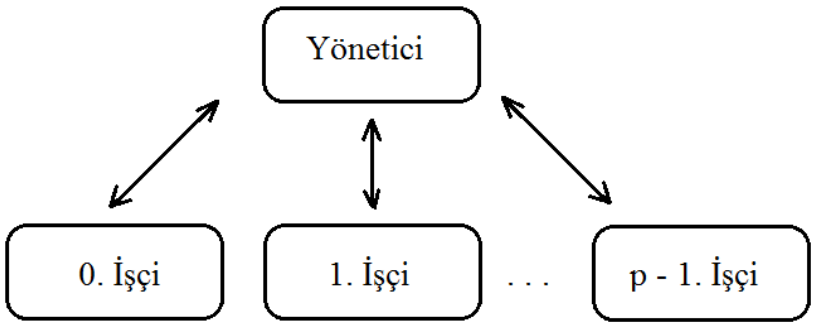
SPMD yaklaşımı alan ayrıştırmasına bir örnektir. Burada tüm süreçlerde aynı kod çalışır.

Bu tarz yaklaşımlar işlemcilerin geniş veri kısımlarında birbirinden bağımsız çalışabileceği sonlu fark algoritmalarında kullanılırlar.

2.3.2. Fonksiyon ayrıştırması

Bazı geniş karmaşık problemlerinde tek başına alan ayrıştırması verimli bir strateji olmayabilir. Örneğin, süreçlere atanmış farklı veri altkümeleri için farklı zamanlar gerektirmesi durumunda kodun performansı en yavaş süreç ile sınırlıdır. Bu durumda **fonksiyon ayrıştırması** (*functional decomposition*) faydalı olur. **Görev paralelleştirilmesi** (*task paralellism*) olarak da bilinen bu yaklaşımda problem çok sayıda alt göreve (fonksiyona) bölünür ve hangi süreç müsait olursa ona atanır.

Yönetici/işçi (*manager/worker*) olarak da bilinen bu yaklaşım bir istemci-sunucu paradigması şeklinde uyarlanır. Bu durum Şekil 2.14.'te resmedilmiştir. Burada bir ana süreç (efendi veya yönetici süreç) işi parçalara böler ve bu parçalar işçi (istemci) süreçlere atanırlar.



Şekil 2.14. Yönetici/işçi modeli

3. MESAJ GEÇİRME ARABİRİMİ

3.1. Giriş

Paralel hesaplama "böl ve yönet" kavramının doğal bir uzantısıdır. İlk olarak elimizde çözmek istediğimiz problemimiz var. Daha sonra problem çözümünde kullanacağımız kaynaklara erişiriz. Son olarak da problemi eş zamanlı çözülebilecek yönetilebilir parçalara ayırıp bu kaynaklara atarız. Bu böl ve yönet stratejisinde zor olan kısım problemi alt problemlere ayırma işi değildir. Esas zorluklar bir problemin paralelleştirilebilen kısımlarını bulmakta ve bilgisayarları paralel çalıştırmaktadır (Karniadakis ve ark., 2003).

Birinci zorluk paralel algoritma geliştirme becerisi arttıkça azalır. Bunun için bol bol örnek çözmek ve farklı algoritmalar tasarlamak gerekmektedir.

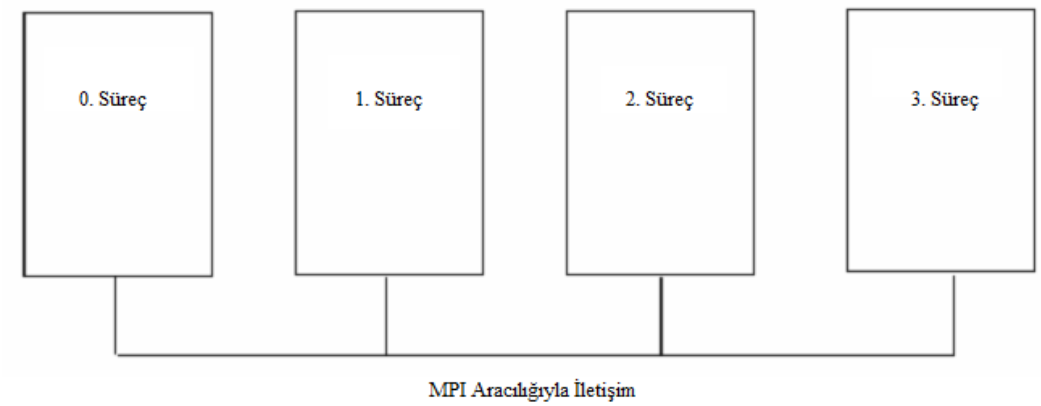
İkincisini kolaylaştırmak için farklı yaklaşımlar geliştirilmiştir. Bu yaklaşımlardan biri farklı bilgisayarlarda çalışan programları birbirleri ile haberleştirerek problemin nihai çözümüne ulaşmaya odaklanır ve mesaj geçirme modeli olarak da bilinir. Bu da farklı bilgisayarlarda eş zamanlı çalışan programların birbirleri ile haberleşebilme kabiliyetine sahip olması demektir.

Bir programın farklı kabiliyetlere sahip olması demek belirli bir problem alanında programın sağlam çözümler üretebilmesi anlamına gelmektedir. Problem alanı çeşit çeşit olduğundan dolayı program geliştiricinin her problemi çözecek fonksiyonlara ve veriyapılarına sahip olması imkansızdır. Bu noktada API (Uygulama Programlama Arabirimi – *Application Programming Interface*) kavramı karşımıza çıkmaktadır. Bir API belli bir problem alanındaki problemleri çözen fonksiyonları, veri yapılarını barındıran ve bu kaynaklara erişecek arabirimleri tanımlayan bir yapıdır. API adreslediği problem alanını soyutlar. Bir API bir veya birden çok programlama dilini destekler ve kendisini kullanan programlara o problem alanında söz sahibi olma imkanı sağlar. Örnek vermek gerekirse, bir araba yarışı oyun programında adreslenmesi gereken çok sayıda fiziksel olgu mevcuttur. Bunun için sadece bu alanda uzmanlaşmış firmalar tarafından geliştirilen ve fiziksel süreçleri modelleyen fizik motorları kullanılır. Bu sayede oyun programları oyununun fiziksel özellikleri anlamında gerçeğe daha yakın bir hal alırlar.

Bir API'nin en önemli özelliği problem alanındaki ayrıntılarla geliştiricileri uğraştırmamasıdır. Geliştirici kendi özel problemine yoğunlaşıp gerisini API'ye bırakır.

Paralel hesaplama programlarında kullanılan mesaj alışverişi ile problem çözmede de aynı yaklaşım kullanılmaktadır. Bilgisayarların birbirleri ile nasıl haberleştiği soyutlanıp program sadece problemin kendisi ile ilgilenir. Gerektiğinde mesaj gönderir veya alır. Arkadaki iletişim bir API aracılığıyla soyutlanmıştır. Programın bunun ayrıntısına girmesine gerek yoktur.

MPI (Mesaj Geçirme Arabirimi – *Message Passing Interface*) paralel çalışan programların haberleşmesini soyutlayan bir uygulama programlama arabiriminin nasıl olması gerektiğini belirleyen bir standarttır. Yüksek başarımlı hesaplama dünyasında çok meşhur olan bu spesifikasyonun esas vaadi şudur: birbirleri ile mesajlar aracılığıyla haberleşerek ortak bir hedefe doğru ilerleyen ve eş zamanlı çalışan süreçlere (veya programlara) imkan vermek. Bu fikir Şekil 3.1.'de dört süreç üzerinden özetlenmiştir (Karniadakis ve ark., 2003):



Şekil 3.1. Birlikte çalışan MPI süreçleri

MPI ne IEEE ne de bir ISO standartıdır. Ancak 40'a yakın kuruluşun katılımıyla geliştirilen ve amacı taşınabilir, verimli ve esnek paralel programların geliştirilmesine bir altyapı sağlamak olan bu spesifikasyon yüksek başarımlı hesaplama camiasında bir "sanayi standartı" haline gelmiştir (Barney, 2010).

MPI spesifikasyonu şöyle özetlenebilir (Gropp ve ark., 2003):

- MPI paralel hesaplamadaki mesaj geçirme modelini adresler. Bu modelde farklı adres uzaylarına sahip süreçler birbirleri ile senkronize olurlar ve mesaj gönderme ve alma mekanizması ile bir sürecin adres uzayından diğerinin adres uzayı arasında veri alışverişi yaparlar.

- MPI bu modeldeki bir API'nin nasıl olması gerektiğini belirleyen bir spesifikasyondur. Bir dil değildir bilakis diğer programlama dillerinden kendi fonksiyonlarına erişim sağlayan mekanizmalar sunar.
- MPI Forum (http://www.mpi-forum.org/) tarafından oluşturulan bu spesifikasyon ilk olarak 1994 yılında MPI-1 olarak ortaya atılmıştır. Daha sonra MPI-2 olarak zenginleştirilen bu standartın son hali MPI-2.2 olarak bilinmektedir. MPI deyince hem MPI-1 hem de MPI-2 standartı anlaşılmaktadır. Bu spesifikasyonun farklı kuruluşlar tarafından geliştirilen ve farklı platformları destekleyen uyarlamarı mevcuttur. Bu tezde kullanılan MPICH2 de MPI-2 standardının Argonne Ulusal Laboratuarları tarafından geliştirilen bir uyarlamasıdır (http://www.mcs.anl.gov/research/projects/mpich2/).

3.1.1. MPI standartı

MPI Forumu tarafından yönetilen ve paralel bilgisayar firmaları, bilgisayar bilimciler ve kullanıcılardan oluşan 40 kadar kuruluşun katılımıyla iki yıl sonucunda geliştirilen MPI-1 standartı 1994 yılında ortaya atılmıştır ve şunlardan oluşur:

- Fortran 77 ve C dillerinden kullanılacak altyordam ve fonksiyonların isimlerini, çağrılma sıralarını ve bunların sonuçlarını belirler. Tüm MPI uyarlamaları bu kurallara uymak zorundadır. Bu sayede taşınabilirlik (*portability*) sağlanmış olur. Bu standartı destekleyen herhangi bir platformda MPI programları derlenebilmeli ve çalıştırılabilmelidir.
- Kütüphanenin ayrıntılı uyarlaması sağlayıcıların, kuruluşların kendine bırakılmıştır. Kendi makinaları için optimize edilmiş versiyonları üretmekte hürdürler.
- MPI-1 standartının farklı platformlarda çalışan uyarlamaları vardır.

Bunun yanında MPI-2 standartı da geliştirilmiştir. Ek özelliklerin yanında bu standartla beraber paralel giriş/çıkış işlemleri sağlanmış, C++ ve Fortran 90 dillerinden çağrılma imkanları sunulmuştur. Artık günümüzdeki tüm MPI uyarlamaları bu standartı desteklemektedirler.

3.1.2. MPI kullanmanın sebepleri

MPI kullanmaya itecek önemli nedenler olarak şunlar sıralanabilir (Barney, 2010):

- Standartlaşma – Standart olarak kabul edilebilecek tek kütüphane MPI kütüphanesidir. Hemen hemen tüm yüksek performanslı hesaplama ortamlarında desteklenmektedir.
- Taşınabilirlik – Uygulamalar MPI standartını destekleyen diğer ortamlara taşındıklarında kaynak kodlarının değiştirlmesine gerek yoktur.
- Performans Fırsatları – Özel veya ticari uyarlamalar alttaki donanımı daha iyi kullanarak performansı optimize edebilirler.
- Fonksiyonellik – Sadece MPI-1'de 115 kadar yordam tanımlanmıştır.
- Bulunabilirlik – Hem ticari alanda hem de açık alanda çok sayıda uyarlama mevcuttur.

Bunların yanında MPI standartının konu edinildiği ve kullanıldığı çok sayıda makale, kitap ve doküman mevcuttur.

3.2. MPI Yordam Çeşitleri

MPI standartında aşağıdaki işlemler için yordamlar tanımlanmıştır (Cyberinfrastructure Tutor, 2011):

- Noktadan noktaya iletişim
- Toplu iletişim
- Süreç grupları
- Süreç topolojileri
- Ortam yönetimi ve sorgusu

3.2.1. MPI program yapısı

Tüm MPI programları aynı genel yapıyı takip etmek zorundadırlar. Bu yapı şöyle özetlenebilir:

MPI başlık dosyasını içer
değişken tanımlamaları

MPI ortamını başlat

...hesaplama ve haberleşme işlemelerini yap...

MPI iletişimlerini kapat

Bu sözde kodun (*pseudocode*) C dilindeki yansıması şöyledir:

```
#include <mpi.h>
int main(int argc, char* argv[])
{
   int rank, boyut;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &boyut);
   // ...hesaplama ve haberleşme işleri...
   MPI_Finalize();
   return 0;
}
```

Yukarıdaki ilkel programda **MPI_Init(int *argc, char ***argv)** metodu MPI ortamını başlatır. Her MPI programının paralel hesaplama anlamında ilk komutu bu olmalıdır. **MPI_Finalize()** metodu da MPI iletişimlerini kapatır. Her MPI programının paralel hesaplama anlamında son komutu bu olmalıdır.

Yukarıdaki kod örneğinde iki fonksiyon daha çağrılmıştır. İlk fonksiyon **rank** değişkenine ilgili programın 0'dan başlamak üzere bulunduğu süreç grubundaki kimlik numarasını vermektedir. Süreç grupları iletişimci objeleri ile temsil edilirler. Bu numaraya **rank** adı verilmektedir. İkincisinin sonucunda ise **boyut** değişkenine bu programı eş zamanlı çalışıtıran bilgisayar sayısı gelir. Bu dört fonksiyon yukarıda bahsedilen ortam yönetimi ve sorgusu grubuna dahildirler.

Bir programın rankını ve kendisinin kaç bilgisayarda eş zamanlı çalıştırıldığını bilmesi işlem yapacağı veri alanını hesaplayabilmesi anlamına gelmektedir. Yukarıda bahsedilen problemi alt parçalara bölme ve SPMD yaklaşımı bu bilgiler ile mümkün olmaktadır.

MPI_COMM_WORLD objesi MPI içinde önceden tanımlanmış ve o oturumdaki tüm MPI süreçlerini temsil eden bir iletişimci objesidir.

3.2.2. MPI veri tipleri

MPI programlarında progralmalama dilinin sağladığı veri tipleri veya MPI tarafından sağlanan önceden tanımlı veri tipleri kullanılabilir. C dilindeki MPI veri

tipleri Çizelge 3.1.'de listelenmişleridir. Bu veri tiplerinin bir avantajı platformdan bağımsız aynı büyüklükteki verileri temsil etmeleridir.

Çizelge 3.1. MPI kütüphanesinin C dilindeki veri tipleri

MPI_CHAR	MPI_UNSIGNED
MPI_SHORT	MPI_FLOAT
MPI_INT	MPI_DOUBLE
MPI_LONG	MPI_LONG_DOUBLE
MPI_UNSIGNED_CHAR	MPI_BYTE
MPI_UNSIGNED_SHORT	MPI_PACKED
MPI_UNSIGNED_LONG	

MPI kütüphanesinin sağladığı ilkel veri tipleri kullanılarak probleme özel veri yapıları tasarlanıp MPI mesaj alışverişinde kullanılabilir. Bu tarz veri türlerine türetilmiş veri türleri denir.

MPI standartının Fortran uyarlamasında karmaşık sayıları destekleyen MPI_COMPLEX, MPI_DOUBLE_COMPLEX gibi veri tipleri de bulunur.

3.2.3. Noktadan noktaya iletişim

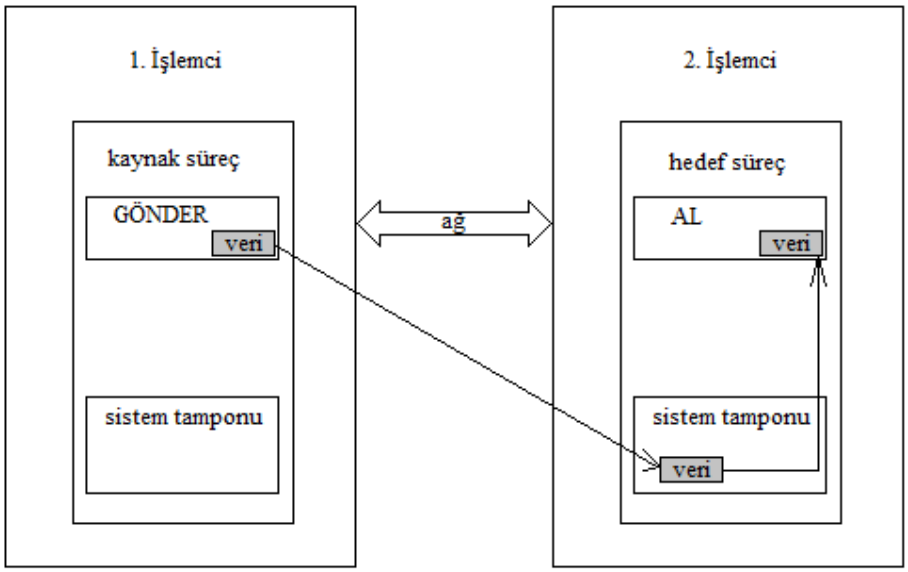
MPI kütüphanesinin sağladığı en temel iletişim imkanı bire-bir iletişim olarak da bilinen noktadan noktaya olan iletişimdir. Kavramsal olarak basit olan iletişim iki süreç arasında olur ve biri mesajı gönderir diğeri de gelen mesajı alır. Pratikte iş bu kadar kolay değildir dikkat edilmesi gereken bazı hususlar vardır. Örnek olarak bir süreç tarafından alınmayı bekleyen mesajlar olabilir. Bu durumda kritik olan MPI ve kabul edecek sürecin hangi mesajı alacağını nasıl belirleyeceğidir. Bir diğer husus da gönderme ve alma yordamlarının iletişimleri başlatıp iletişimin sonucunu beklemeden hemen sonlanmaları veya başlatılan iletişimin bitmesini bekleyip bundan sonra sonlanmaları mevzusudur.

3.2.3.1. Kaynak ve hedef

Noktadan noktaya iletişim iki taraflı bir iletişimdir. Her iki taraftaki süreçlerin aktif katılımını gerektirir. Bunlardan biri kaynak olup gönderen taraftır. Diğeri de hedef görevini yerine getirir ve alan taraftır.

Genellikle kaynak ve hedef süreçler asenkron bir şekilde çalışırlar. Bir tek mesajın gönderimi ve alımı bile senkronize edilmez. Hedef süreç mesajı almaya

başlamadan çok önce kaynak süreç mesajı göndermiş olabilir veya hedef süreç henüz gönderilmemiş bir mesajı alma işlemini başlatabilir. Birinci durumu yönetmek için ilgili MPI uyarlamasının bir tampon mekanizması sağlaması gerekmektedir. Bu durum Şekil 3.2.'de özetlenmeye çalışılmıştır. Bunun için genellikle sistem tamponu kullanılır (Barney, 2010).



Şekil 3.2. Hedef süreçte tamponlanmış mesajın yolu

3.2.3.2. Mesajlar

Zarf ve mesaj gövdesi olmak üzere mesajlar iki kısımdan oluşurlar. Zarfın dört bileşeni vardır:

- Kaynak – gönderen süreç.
- Hedef – alan süreç.
- İletişimci – kaynak ve hedefin de ait olduğu süreçler kümesi.
- Etiket – mesajları sınıflamak için kullanılan bileşen.

Mesaj gövdesi ise üç parçadan oluşur:

- Tampon – mesajın gerçek verisi.
- Veri tipi – mesaj verisinin türü.
- Sayısı – veri tipi türünde tampondaki veri sayısı.

Bir tampon bir vektör olarak düşünülebilir. Veri tipleri ve sayılar kullanılarak temel veri tipleri dışındaki verilerin de gönderilmesi sağlanabilir.

3.2.3.3. Mesajların gönderilmesi ve alınması

Mesajın gönderilmesi sırasında kaynak gizliden belliyken mesajın geri kalan kısmı gönderen süreç tarafından açık bir şekilde belirtilir. Gönderme ve alma işlemleri senkronize edilmediğinden dolayı süreçlerin göndermiş olduğu ama henüz alınmamış birden çok mesajları olabilir. Bu mesajlara **bekleyen mesajlar** adı verilir. Bekleyen mesajların basit bir FIFO (İlk gelen ilk çıkar – *First In First Out*) yaklaşımı ile işlenmemesi MPI'ın en önemli özelliklerinden biridir. Her bekleyen mesajın birden çok özelliği vardır ve hedef süreç hangi mesajı alacağını bu özelliklere göre belirler.

Hedef süreç bir zarf belirler ve bekleyen mesajlar içinde bu zarfa uyan mesajları alır. Zarfa uyan bir mesaj olmadığı sürece alma işlemi tamamlanmamış olarak bekler.

3.2.3.4. İletişim modları ve tamamlanma kriterleri

İletişim modları mesajları iletmek için kullanılan prosedürleri ve iletişimin ne zaman tamamlandığını belirleyen kriterleri tanımlarlar. Gönderme işlemi için dört iletişim modu mevcuttur:

- Standart
- Senkron
- Tamponlu
- Hazır

Senkron gönderimde hedef mesajla ilgili bilgilendirildiği anda gönderim tamamlanır. Tamponlu gönderimde ise giden veriler yerel bir tampona kopyalandıkları zaman gönderim tamamlanır.

Alma işleminde ise tek iletişim modu vardır. Bir alma işlemi gelen veri tam olarak ulaştığında ve kullanıma hazır olduğunda tamamlanır.

3.2.3.5. Tıkanmalı ve tıkanmasız iletişim

Bir gönderme veya alma işlemi tıkanmalı (*blocking*) ve tıkanmasız (*nonblocking*) olmak üzere ikiye ayrılır.

Tıkanmalı iletişimde gönderme veya alma metodu ilgili işlem tamamlanmadan sonlanmaz. Program işlem bitene kadar ilgili fonksiyonda bekler. Gönderen süreçte işlemden sonra gönderilen değişkenlere farklı değerler atanabilir. Alan süreç de işlem sonucunda verinin tamamının geldiğinden emindir.

Tıkanmasız iletişimde ise program gönderme veya alma metodunu çağırdıktan sonra ilgili satırdan sonraki ifadelerini gerçekleştirmeye devam eder. Tamamlanma kriterinin sağlanıp sağlanmadığından haberi olmaz. Bu yaklaşımın en önemli avantajı arkaplanda iletişim sürerken buna takılıp kalmadan başka bir hesapla uğraşabilir. Daha sonra iletişimin tamamlanıp tamamlanmadığını test eder ve tamamlanması durumunda iletişim sonrası lojiğine devam eder. Tamamlanmadıysa diğer hesapları yapmaya devam eder.

3.2.3.5. Tıkanmalı gönderme ve alma

Tıkanmalı gönderme metodunun C dilindeki deklarasyonu şöyledir:

```
int MPI_Send(  void*          tampon,
               int            sayi,
               MPI_Datatype   veritipi,
               int            hedef,
               int            etiket,
               MPI_Comm       iletisimci);
```

Tıkanmalı alma metodunun ilgili deklarasyonu şöyledir:

```
int MPI_Recv(  void*          tampon,
               int            sayi,
               MPI_Datatype   veritipi,
               int            kaynak,
               int            etiket,
               MPI_Comm       iletisimici,
               MPI_Status*    durum);
```

Burada gönderme metodundaki **tampon**, **sayi** ve **veritipi** değişkenleri mesaj gövdesini; **hedef**, **etiket** ve **iletisimci** değişkenleri ise mesajın zarfını temsil ederler.

Alma metodundaki **tampon**, **sayi** ve **veritipi** değişkenleri mesaj gövdesini; **kaynak**, **etiket** ve **iletisimci** değişkenleri ise mesajın zarfını temsil ederler. Herhangi bir kaynaktan veya herhangi bir etikete sahip mesajları almak için eşleme (*wildcard*) değerleri kullanılabilir. Herhangi bir kaynaktan gelen mesajı almak için **MPI_ANY_SOURCE** eşleme değeri ve herhangi bir etikete sahip mesajı almak için de

MPI_ANY_TAG eşleme değeri kullanılır. Burada da **durum** değişkeni devreye girmektedir. Bu değişken fonksiyonun sonucunda değer alır ve alınan mesajın durumu hakkında bazı bilgileri içerir. **durum.MPI_SOURCE** ve **durum.MPI_TAG** özellikleri sırasıyla mesajın kaynağını ve etiketini gösterirler.

Gönderici ve alıcı süreçler verinin veri tipi konusunda anlaşmalıdırlar, aksi halde iletişim gerçekleşmez ve program takılır. Bu anlaşmayı sağlamak programı yazanın görevidir.

3.2.3.6. Tıkanmalı gönderme ve alma örnek uygulaması

Aşağıdaki örnek programda gönderme ve alma fonksiyonları paralel çalışan iki süreç üzerinden gösterilmiştir. Daha doğrusu, aşağıdaki program iki süreç ile çalıştırıldığı zaman anlamlı bir iş yapmaktadır. Programa göre sıfırıncı süreç **k[10]** dizisini oluşturup bunu birinci süreçteki **h[10]** dizisine gönderir. Hedef süreç de kaynak süreçten veri alır. Program içindeki akış **rank** değişkenine göre belirlenir. Sözkonusu program şöyledir:

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>

int main (int argc, char **argv)
{
   int rank,i;
   int etiket = 111;
   MPI_Status durum;
   double k[10],h[10];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if( rank == 0 )
   {
      for (i=0;i<10;++i)
         k[i]=sqrt((double)i);
      MPI_Send(k, 10, MPI_DOUBLE, 1, etiket, MPI_COMM_WORLD);
   }
   else if( rank == 1 )
   {
      MPI_Recv(h, 10, MPI_DOUBLE, 0, etiket, MPI_COMM_WORLD, &durum);

   }
   MPI_Finalize();
   return 0;
}
```

3.2.3.7. Kilitlenme

MPI_Send ve **MPI_Recv** fonksiyonlarını çağıran süreçler bu fonksiyonlar sonlanana kadar beklerler. İlgili iletişim tamamlanana kadar bu bekleme devam eder. Eğer biri diğerinin ilerlemesini bekliyorsa kilitlenme (*deadlock*) durumu oluşur. Bu durumda hiçbir süreç ilerlemez. Aşağıda program örneği bu durumu göstermektedir:

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv)
{
   int rank,i;
   int eti_kay_hed = 111;
   int eti_hed_kay = 112;
   MPI_Status durum;
   double k[10], h[10];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if( rank == 0 )
   {
      // once al, sonra gonder
      MPI_Recv(h, 10, MPI_DOUBLE, 1, eti_hed_kay, MPI_COMM_WORLD, &durum);
      MPI_Send(k, 10, MPI_DOUBLE, 1, eti_kay_hed, MPI_COMM_WORLD);
   }
   else if( rank == 1 )
   {
      // once al, sonra gonder
      MPI_Recv(h, 10, MPI_DOUBLE, 0, eti_kay_hed, MPI_COMM_WORLD, &durum);
      MPI_Send(k, 10, MPI_DOUBLE, 0, eti_hed_kay, MPI_COMM_WORLD);
   }
   MPI_Finalize();
   return 0;
}
```

Bu programda ne mesaj gönderilir ne de alınır. 0. süreç 1. süreç bir mesaj göndermediği sürece ilerlemez. 1. süreç de 0. süreç bir mesaj göndermeden ilerlemez. Kilitlenme olmuştur.

Kilitlenmeyi önlemek programın iletişim organizasyonunun düzgün kurulmasına bağlıdır. Yukarıdaki programda **else if** bloğundaki **MPI_Send** ve **MPI_Recv** fonksiyonları yer değiştirilirse kilitlenme durumu ortadan kalkar.

3.2.3.8. Tıkanmasız gönderme ve alma

Tıkanmalı iletişimde gönderme ve alma fonksiyonları ilgili iletişim bitene kadar takılıp kalırlar. Bu bazen gecikmelere hatta kilitlenmelere sebep olabilir. Bundan dolayı MPI ortamında çağıran süreçleri tıkamayan gönderme ve alma fonksiyonları sağlanmıştır. Bunun için iki ikişer fonksiyon kullanılır. Bunlardan ilki iletişimi başlatır diğeri de sonlandırır. Bu iki çağrı arasında programlar kendilerini diğer hesaplara verebilirler. Bu tarz iletişime **tıkanmasız iletişim** denir. Arka planda gerçekleşen iletişimde herhangi bir farklılık sözkonusu değildir. Esas farklılık iletişimin kaynak ve hedefinde olabilir.

Bu tarz iletişimde gönderme veya alma işi başladıktan sonra ilgili işleme bir referans gerekir. Bu referans sayesinde işlemin durumu takip edilir veya bitmesi beklenir. Bunun için istek tanıtıcı değeri (*request handle*) denilen yapılar kullanılır. Bunlar MPI ortamında **MPI_Request** veri tipi ile temsil edilirler.

Tıkanmasız göndermeyi başlatmak için

```
int MPI_Isend( void*        tampon,
               int          sayi,
               MPI_Datatype veritipi,
               int          hedef,
               int          etiket,
               MPI_Comm     iletisimci,
               MPI_Request* tanitici);
```

fonksiyonu kullanılır. Bu fonksiyonda ilk üç argüman mesajın gövdesini diğe üçü de zarfını tanımlar. Mesaj kaynağı örtük tanımlanır. Fonskiyondan çıkınca gönderme başlamış ama bitmemiştir. Tamamlamak için diğer bir fonksiyonun çağrılması gereklidir. Bir diğer önemli nokta da iletişim tamamlanıncaya kadar bu fonksiyona geçilen herhangi bir argüman ne okunmalı ne de üzerine veri yazılmalıdır.

Tıkanmasız almayı başlatmak için

```
int MPI_Irecv( void*        tampon,
               int          sayi,
               MPI_Datatype veritipi,
               int          kaynak,
               int          etiket,
               MPI_Comm     iletisimici,
               MPI_Request* tanitici);
```

fonksiyonu kullanılır. Burada **MPI_Recv** fonksiyonundaki **MPI_Status** değişkeninin yerini **MPI_Request** değişkeni alır. **MPI_Isend** fonksiyonu için yazılanlar bu fonksiyon için de geçerlidir. Tekrar etmekte faydası olan bir husus gönderme

tamamlanıncaya kadar argümanlara herhangi bir yazma veya okuma işinin yapılmaması gerektiği hususudur.

Başlatılan iletişimin tamamlanması gerekmektedir. Tıkanmasız iletişimde tamamlanma durumu tıkanmalı ve tıkanmasız olmak üzere iki farklı fonksiyondan biri ile kontrol edilebilir. Tıkanmalı kontrol fonksiyonları **MPI_Wait** ve bunun varyansyonlarıdır. Tıkanmasız kontrol fonksyionları ise **MPI_Test** ve bunun varyasyonlarıdır.

Tıkanmalı kontrol fonksiyonunun ayrıntısı şöyledir:

```
int MPI_Wait(   MPI_Request*   tanitici,
                MPI_Status*    durum);
```

tanitici değişkeni bir önceki başlatılan gönderme veya alma işlemini temsil eder. İletişim tamamlanınca bu fonksiyon döner. Eğer alma işlemi başlatıldı ise verinin hangi kaynaktan geldiği, hangi etiketi kullandığı ve gelen verinin sayısı **durum** değişkeni aracılığıyla bildirilir. Gönderme işlemi başlatıldı ve hata olduysa durum değişkeninde bu hatayı temsil eden kod bulunur.

Tıkanmasız kontrol fonksiyonunun ayrıntısı şöyledir:

```
int MPI_Test(   MPI_Request*   tanitici,
                int*           bayrak,
                MPI_Status*    durum);
```

Burada da **tanitici** değişkeni bir önceki başlatılan gönderme veya alma işlemini temsil eder. Bu fonksiyon iletişim tamamlanmasını beklemeden hemen döner. Bu durumda **bayrak** değişkenine bakılır. Eğer bunda "doğru" değeri varsa başlatılan iletişim tamamlanmıştır. Eğer "yanlış" değeri varsa iletişim tamamlanmamıştır. İletişim sürecindeki boş zamanlarda programları başka işler için kullanmak da bu değişken sayesinde yapılır. Örneğin gönderme sırasındaki boş zamanının başka işler için kullanımı şöyle bir sözde kod ile temsil edilebilir:

Tıkanmasız gönderme işlemini başlat
MPI_Test fonksiyonunu çağır sonucu *bayrak* değişkenine yaz
***bayrak* değişkeni yanlış olduğu sürece şunları yap:**
    İletişim dışındaki diğer işleri yap
    MPI_Test fonksiyonunu çağır sonucu *bayrak* değişkenine yaz
İletişim sonrası işlere devam et

Tamamlama kriteri olarak bu fonksiyonun kullanılması kilitlenmelerin oluşma ihtimalini azaltır ancak kod karmaşıklığını arttırdığı için koddaki hataların ayıklanması ve kodun bakımı zorlaşabilir. Genel olarak gecikmelerin fazla olduğu sistemlerde tıkanmasız tamamlama kontrolünün yapılması tavsiye edilmektedir.

3.2.3.9. Tıkanmasız alma örnek uygulaması

Kitilenme konusunda gösterilen örnekte alma işi için tıkanmasız alma yöntemi kullanılırsa kilitlenme olmaz. Aşağıda bu kodun örneği verilmiştir:

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv)
{
   int rank,i;
   int eti_kay_hed = 111;
   int eti_hed_kay = 112;
   MPI_Request tanitici;
   MPI_Status durum;
   double k[10], h[10];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if( rank == 0 )
   {
      /* almayı başlat, gönder ve bekle */
      MPI_Irecv(h, 10, MPI_DOUBLE, 1, eti_hed_kay, MPI_COMM_WORLD, &tanitici);
      MPI_Send(k, 10, MPI_DOUBLE, 1, eti_kay_hed, MPI_COMM_WORLD);
      MPI_Wait(&tanitici, &durum);
   }
   else if( rank == 1 )
   {
      /* almayı başlat, gönder ve bekle */
      MPI_Irecv(h, 10, MPI_DOUBLE, 0, eti_kay_hed, MPI_COMM_WORLD, &tanitici);
      MPI_Send(k, 10, MPI_DOUBLE, 0, eti_hed_kay, MPI_COMM_WORLD);
      MPI_Wait(&tanitici, &durum);
   }
   MPI_Finalize();
   return 0;
}
```

Bu kod örneğinde her iki süreçte kod göndermeye geldiği zaman takılır. Ama bu sefer gönderme başarılı olur çünkü buna uygun bir alma başlatılmış durumdadır. Dolayısıyla kilitlenme olmaz.

3.2.3.10. Gönderme ve alma modları

MPI kütüphanesinde dört adet gönderme ve bir adet alma modu mevcuttur. Gönderme modları şunlardır:

- Standart Gönderme Modu
- Senkron Gönderme Modu
- Hazır Gönderme Modu
- Tamponlu Gönderme Modu

Mesaj hangi modda gönderilirse gönderilsin alma süreci MPI_Recv veya MPI_Irecv fonksiyonlarından birini kullanarak bu mesajı alabilir.

Her dört gönderme modunun tıkanmalı ve tıkanmasız olmak üzere ikişer fonksiyonu vardır. Toplam 8 adet fonksiyon Çizelge 3.2.'de gösterilmiştir. **Tıkanmalı Fonksiyonlar** sütunundaki **MPI_Send** dışındaki diğer fonksiyonlar bu fonksiyonun aldığı argümanları aynı sırada alırlar. **Tıkanmasız Fonksiyonlar** sütunundaki **MPI_ISend** dışındaki diğer üç fonksiyon bu fonksiyonun aldığı argümanları aynı sırada alırlar.

Çizelge 3.2. MPI ortamındaki gönderme fonksiyonları

Gönderme Modu	Tıkanmalı Fonksiyon	Tıkanmasız Fonksiyon
Standart	MPI_Send	MPI_Isend
Senkron	MPI_Ssend	MPI_Issend
Hazır	MPI_Rsend	MPI_Irsend
Tamponlu	MPI_Bsend	MPI_Ibsend

Standart gönderme modu (*standart mode send*) genel amaçlı bir gönderme modudur. Diğer üçü özel durumlarda kullanışlı olabilirler. Standart gönderme modunda bir gönderme MPI tarafından çalıştırıldığı zaman ya mesaj bir MPI iç tampon alanına kopyalanıp asenkron bir şekilde hedef sürece transfer edilir ya da kaynak ve hedef süreçler mesaj üzerinde senkronize olurlar. Mesajın boyutu, eldeki kaynaklar ve diğer faktörleri dikkate alan MPI uyarlaması bu iki mod içinden birini seçme konusunda özgürdür. Eğer iç tampon alanına kopyalama ile gönderme olacaksa, bu kopyalama biter bitmez gönderme tamamlanır. Eğer iki süreç senkronize olurlarsa alan süreç göndermeye uyan (zarfı eşleşen) bir alma başlatıp mesajı almaya başladığı anda gönderme tamamlanır. Bu durum bu moddaki hem tıkanmalı hem tıkanmasız

fonksiyonlar için geçerlidir. Dikkat edilmesi gereken önemli bir husus MPI_Isend fonksiyonu ile gönderme başlatıldığı zaman ilgili işlem tamamlanana kadar bu fonksiyona geçilen parametreler ile herhangi bir işlem yapılmamasıdır. Bu kuralın ihlali programın çökmesine sebep olabilir.

Bu gönderme modunun en önemli avantajı gönderme işleminin tamponlu mu yoksa senkron mu olacağının MPI ortamına bırakılmasıdır. Bu sayede performans elde etme ihtimali artar.

Senkron gönderme modunda (*synchronous mode send*) gönderme işlemi iki süreci senkronizasyonu sonucu yapılır. Hedef süreç mesajı almaya başladığı zaman gönderme tamamlanır.

Hazır gönderme modunda (*ready mode send*) gönderilme işlemi başlatılmadan buna uyacak bir alma işleminin başlatılmış olması gereklidir. Böyle bir alma başlatılmadı ise sonuç tanımsızdır. Hedef sürecin hazır olma durumu bilinebiliyorsa bu modun kullanımı iletişim hızı açısından avantaj sağlayabilir.

Tamponlu gönderme modunda (*buffered mode send*) yukarıda bahsedilen tampon alanları kullanılır. Bu durumda da her zaman yeteri kadar tampon olacağı varsayımı sözkonusudur. Gönderme işlemini tamamlayacak kadar tampon alanı yoksa sonuç tanımsızdır.

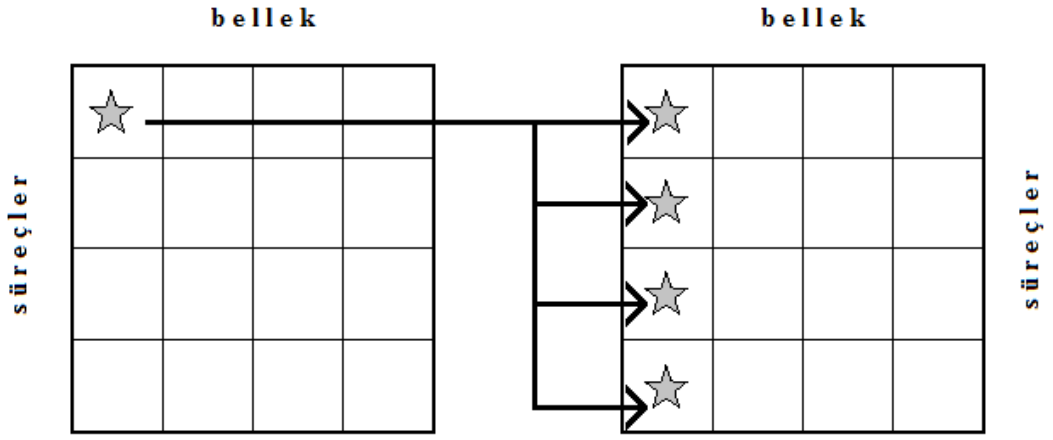
3.2.4. Toplu iletişim

İkiden çok süreç arasındaki veri iletişimine toplu iletişim denir. Süreçler arasındaki tüm veri hareketleri noktadan noktaya iletişim fonksiyonları kullanılarak yapılabilir. Bazı iletişim işlemleri çok sık olduğundan dolayı bunlar için toplu iletişim fonksiyonları sağlanmıştır. Bu fonksiyonlar arka planda noktadan noktaya iletişim fonksiyonlarını kullanırlar. Bunların üzerine bir soyutlama sözkonusudur.

Toplu iletişimde bir gruptaki tüm süreçler iletişime katılırlar. Burada alma ve göndermeleri eşlemek için etiketleme mekanizması kullanılmaz. Bu eşleme programın çalışma sırasına göre sağlanır ve bu hususta dikkatli olunmalıdır.

3.2.4.1. Yayımlama işlemleri

Toplu işlemler içinde en basit olanı yayımlama (*broadcast*) işlemidir. Bu işlemde bir süreç gruptaki diğer süreçlere veri gönderir. Şekil 3.3.'te bu işlem dört süreç üzerinden resmedilmiştir.



Şekil 3.3. Yayımlama işlemi

Bu işlem için MPI kütüphanesindeki

```
int MPI_Bcast( void*          tampon,
               int            sayi,
               MPI_Datatype verituru,
               int            kok,
               MPI_Comm       iletisimci);
```

fonksiyonu kullanılır. Burada **kok** parametresi yayım yapan kök süreci temsil eder. MPI_Bcast fonksiyonu veriyi iletişimcideki tüm süreçlere gönderir. Aynı işlem kök süreçten her sürece gönderme işlemi başlatılarak da yapılabilir, ama bu fonksiyon veri hareketini daha kolay yapar. Bu fonksiyon her süreçte çağrılmalıdır.

Aşağıdaki programda 0. süreç kök görevi görmekte ve tüm süreçlere (**MPI_COMM_WORLD** parametresi sayesinde) **parametre** değişkenindeki değeri göndermektedir:

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char *argv[])
{
   int rank;
   double parametre;
   MPI_Init(&argc, &argv);
```

```
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if(rank==0)
      parametre=3.14;
   MPI_Bcast(&parametre,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
   printf("P:%d yayımlamadan sonra parametre değeri %f\n",rank,parametre);
   MPI_Finalize();
}
```

Bu program 4 süreç üzerinde çalıştırıldığında şöyle bir çıktı elde edilir:

```
P:0 yayımlamadan sonra parametre değeri 3.14
P:1 yayımlamadan sonra parametre değeri 3.14
P:2 yayımlamadan sonra parametre değeri 3.14
P:3 yayımlamadan sonra parametre değeri 3.14
```

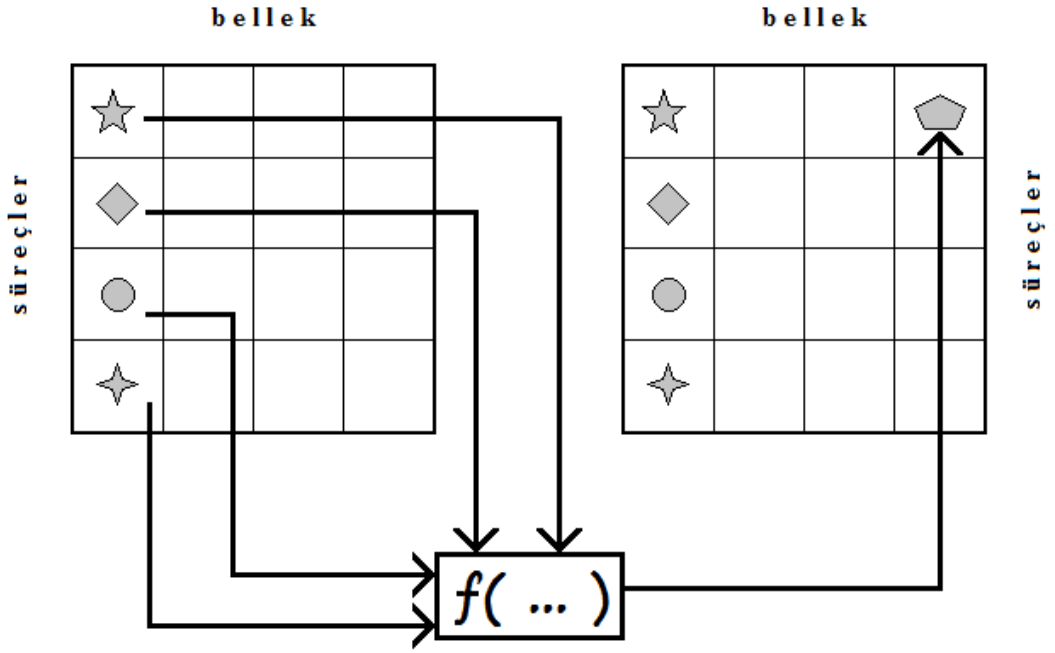
3.2.4.2. İndirgeme işlemleri

Bir sürecin (kök süreç) gruptaki diğer süreçlerden veri toplayıp bu veriler üzerinde işlem yapıp kendisine almasını sağlayan işlemlere indirgeme (*reduction*) işlemleri denir. Bu tüm süreçlere dağılmış bir dizinin elemanlarını toplamak olabileceği gibi bu dizi elemanları arasındaki en küçük veya en büyük değerin bulunması da olabilir. Bu işlem Şekil 3.4.'te dört süreç üzerinden gösterilmiştir. Şekildeki dört farklı süreçteki farklı değerlerin alınıp bir fonksiyondan geçirildikten sonra kök süreçteki bir bellek bölgesine yazılması gösterilmiştir. İlk satırdaki süreç kök süreç olarak gösterilmiştir.

Bu işlem için MPI kütüphanesindeki

```
int MPI_Reduce(void*          gonderme_tamponu,
               void*          alma_tamponu
               int            sayi,
               MPI_Datatype   verituru,
               MPI_Op         islem,
               int            kok,
               MPI_Comm       iletisimci);
```

fonksiyonu kullanılır.



Şekil 3.4. İndirgeme işlemi

MPI_Reduce fonksiyonu **gonderme_tamponu** parametresi ile temsil edilen yerden değer alır, **islem** parametresi ile temsil edilen işlemi uygular **alma_tamponu** ile temsil edilen kök sürecin belleğine yazar. Çizelge 3.3.'te **MPI_Op** olarak kullanılabilecek değerler ve bunların açıklaması verilmiştir.

Çizlege 3.3. MPI_Reduce fonksiyonu ile kullanılabilecek işlemler ve açıklamaları

İşlem	Açıklaması
MPI_MAX	Maksimum (*maximum*)
MPI_MIN	Minimum (*minimum*)
MPI_SUM	Toplam (*sum*)
MPI_PROD	Çarpım (*product*)
MPI_LAND	Mantıksal ve (*logical and*)
MPI_BAND	Bit seviyesinde ve (*bit-wise and*)
MPI_LOR	Mantıksal veya (*logical or*)
MPI_BOR	Bit serviyesinde veya (*bit-wise or*)
MPI_LXOR	Mantıksal özel veya (*logical exclusive or*)
MPI_BXOR	Bit seviyesinde özel veya (*bit-wise exclusive or*)
MPI_MINLOC	Global minimum değeri ve bu minimuma eklenmiş gelen indisi hesaplar
MPI_MAXLOC	Global maksimum değeri ve bu maksimuma eklenmiş gelen indisi hesaplar

Aşağıdaki MPI programı bu fonksiyonun kullanımını göstermektedir. Bu programda kök süreç olarak rank değeri 7 olan süreç kullanılmıştır ve sonuç da bu süreçten yazılmaktadır. Program 8 adet süreç üzerinden çalıştırıldığı zaman 1'den 8'e kadar olan sayıların çarpımını hesaplar.

Programda kullanılan ve daha önce bahsedilmemiş MPI fonksiyonlarından biri de **MPI_Barrier(MPI_Comm iletisimci)** fonksiyonudur. Bu fonksiyonun görevi gruptaki tüm üyeler bu fonksiyonu çağırana kadar çağrıldığı yerde çağıranı beklemeye almaktır. Bu fonksiyondan çıkıldığı anda gruptaki tüm süreçler bu noktada senkronize olmuşlardır. Kök sürecin dosya okuduğu ve diğer süreçlere gönderdiği işlemlerde bu senkronizasyon gereklidir. Giriş/çıkış işlemi bitince diğer süreçlerin hesaba katılmaları gerekir. Bu programdaki kullanılma sebebi ise farklı işlemcilerdeki olası gecikmelerden dolayı eğer **MPI_Reduce** fonksiyonuna gelindiğinde bir veya birden fazla işlemcide **kaynak** değişkenine değer atanmadıysa sonuç belirsiz olur.

```
#include   <stdio.h>
#include   <mpi.h>
void main(int argc, char *argv[])
{
   int rank;
   int kaynak,sonuc,kok;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   kok=7;
   kaynak=rank+1;
   MPI_Barrier(MPI_COMM_WORLD);
   MPI_Reduce(&kaynak,&sonuc,1,MPI_INT,MPI_PROD,kok,MPI_COMM_WORLD);
   if(rank==kok)
      printf("P:%d MPI_PROD sonucu %d \n",rank,sonuc);
   MPI_Finalize();
}
```

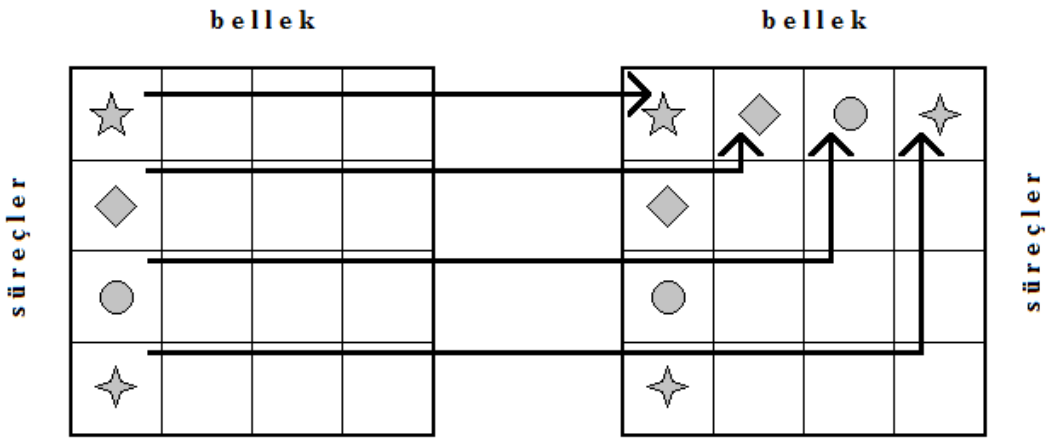
3.2.4.3. Bir araya getirme işlemleri

Bu işlemler gruptaki işlemcilere dağılmış verileri toplar ve bir işlemcide düzgün bir sırada bir araya getirirler. Bu gruba giren iki adet fonksiyon mevcuttur. Bunlardan ilki olan ve tanımı

```
int MPI_Gather(void          *gonderme_tamponu,
          int               gonderme_sayisi,
          MPI_Datatype      gonderme_veritipi,
          void              *alma_tamponu,
          int               alma_sayisi,
          MPI_Datatype      alma_veritipi,
          int               kok,
          MPI_Comm          iletisimci);
```

şeklinde olan fonksiyon hepsinden bire (*all-to-one*) türünde bir fonskiyondur. Alma argümanları sadece kök süreçte bir anlam ifade ederler. Bu fonksiyon çağrıldığı zaman

iletişimcideki tüm süreçler gönderme tamponunda bulunan verileri kök sürece gönderirler. Kök süreç bunları aldığı zaman rank sırasına göre saklar. Bu fonksiyon dört süreç üzerinden Şekil 3.5'te gösterilmiştir.



Şekil 3.5. MPI_Gather fonksiyonunun gösterimi

Yukarıdaki şekilde dört süreçten gelen veriler 0. sürecin belleğine ranklarına göre sıralanarak yazılmışlardır.

Bu fonksiyonun önşartlarından biri tüm süreçlerin aynı miktarda veri göndermeleri gerekliliğidir.

Aşağıdaki örnek programda bu fonksiyonun kullanımı gösterilmiştir:

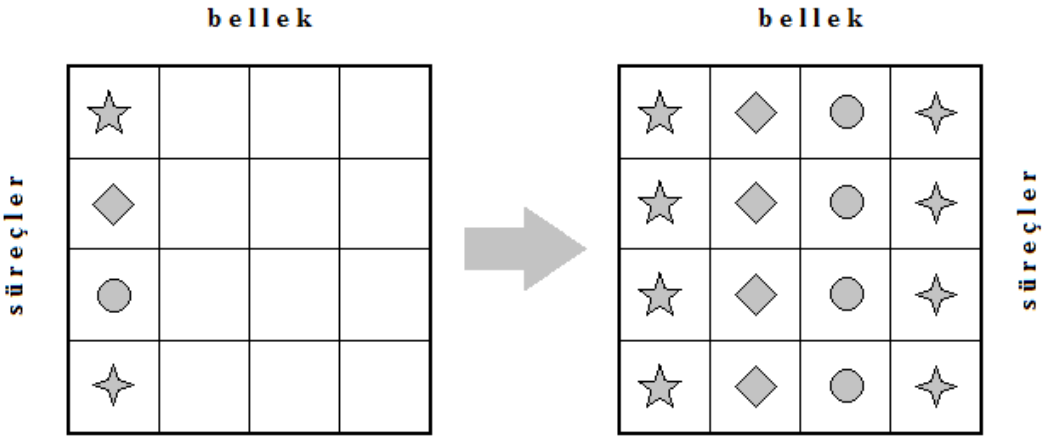
```
#include   <stdio.h>
#include   <mpi.h>
void main(int argc, char *argv[])
{
   int rank,boyut;
   double alma_tamponu[10],gonderme_tamponu;
   int gonderme_sayisi,alma_sayisi;
   int i;
   int kok = 3;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&boyut);
   gonderme_sayisi=1;
   gonderme_tamponu=10.0+rank;
   if(rank==kok)
   {
      alma_sayisi=1;
   }
   MPI_Gather(&gonderme_tamponu,gonderme_sayisi,MPI_DOUBLE,
      alma_tamponu,alma_sayisi,MPI_DOUBLE,kok,MPI_COMM_WORLD);
   if(rank==kok)
   {
      for(i=0;i<boyut;++i)
```

```
    {
      printf("P:%d alma_tamponu[%d] = %f \n",rank,i,alma_tamponu[i]);
    }
  }
  MPI_Finalize();
}
```

Bu programda tüm süreçlerdeki oluşturulan **gonderme_tamponu** değerleri kök süreçteki **alma_tamponu** dizisine sıralı bir şekilde alınıp kök süreçten ekrana çıktı olarak yazılmaktadır. Bu program dört süreç üzerinden çalıştırıldığı zaman şöyle bir sonuç ekrana yansır oluşur (kök süreç 3. süreçtir):

```
P:3 alma_tamponu[0] = 10.00
P:3 alma_tamponu[1] = 11.00
P:3 alma_tamponu[2] = 12.00
P:3 alma_tamponu[3] = 13.00
```

Tanımı kok parametresi hariç **MPI_Gather** ile aynı olan bir diğer bir araya getirme fonksiyonu da **MPI_Allgather** fonksiyonudur. Bu fonksiyonun öncesi ve sonrası Şekil 3.6'da gösterilmiştir. Bu fonksiyon süreçlerden topladığı verileri ranklarına göre sıralayıp tüm süreçlerin alma tamponlarına yazar.



Şekil 3.6. MPI_Allgather fonskiyonunun gösterimi

Bu fonksiyonu kullanmadan bu işi yapmak için önce kök süreçte **MPI_Gather** fonksiyonu çağrılır daha sonra tüm diğer süreçlere **MPI_Bcast** ile yayımlanır. Ama **MPI_Allgather** daha verimli sonuçlar verir.

Aşağıdaki programda bu fonksiyonun kullanımı gösterilmiştir. Programın bir önceki programdan farkı kök sürecin önemsiz olması ve söz konusu fonksiyondan

dolayı verilerin tüm süreçlere gelmesidir. Bunu göstermek için de tüm süreçlerden çıktı alınmaktadır.

```
#include   <stdio.h>
#include   <mpi.h>
void main(int argc, char *argv[])
{
   int rank,boyut;
   double alma_tamponu[10],gonderme_tamponu;
   int gonderme_sayisi,alma_sayisi;
   int i;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&boyut);
   alma_sayisi = gonderme_sayisi=1;
   gonderme_tamponu=10.0+rank;
   MPI_Allgather(&gonderme_tamponu,gonderme_sayisi,MPI_DOUBLE,
      alma_tamponu,alma_sayisi,MPI_DOUBLE,MPI_COMM_WORLD);
   for(i=0;i<boyut;++i)
   {
      printf("P:%d alma_tamponu[%d] = %f \n",rank,i,alma_tamponu[i]);
   }
   MPI_Finalize();
}
```

Her süreç 10.0 değerine kendi rankını ekleyip bu değerler kök süreçte toplanmaktadırlar.

3.2.4.4. Saçılma işlemleri

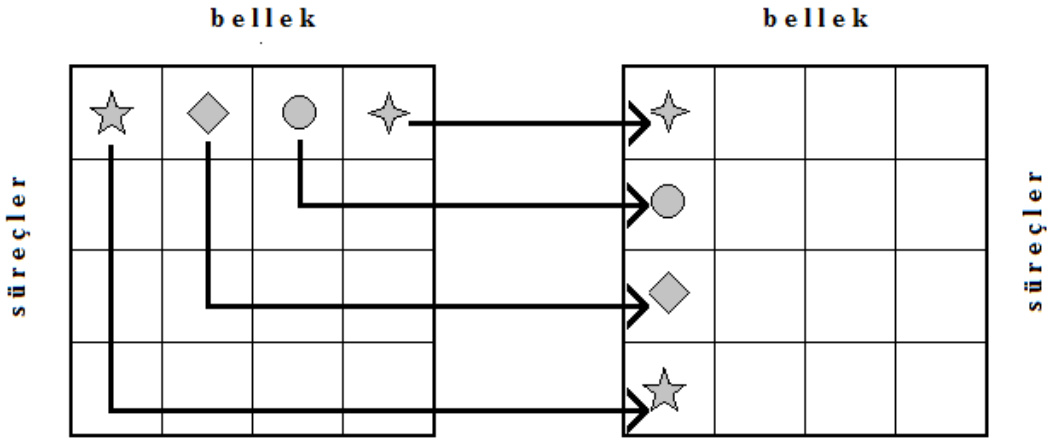
MPI_Gather fonksiyonunun tersi olan bir toplu iletişim biçimi vardır. Bu da saçılma (*scatter*) işlemidir. Bu bir birden hepsine (*one-to-all*) iletişim biçimidir; kök süreçten diğer bütün süreçlere rank sırasına göre farklı veriler gönderilir.

MPI kütüphanesinde bu işlem için

```
int MPI_Scatter(void      *gonderme_tamponu,
          int             gonderme_sayisi,
          MPI_Datatype    gonderme_veritipi,
          void            *alma_tamponu,
          int             alma_sayisi,
          MPI_Datatype    alma_veritipi,
          int             kok,
          MPI_Comm        iletisimci);
```

fonksiyonu kullanılır.

Bu fonksiyon çağrıldığı zaman kök süreç **gonderme_tamponu** ile temsil edilen verileri eşit parçalara böler ve her parçayı bir sürece gönderir. Bunun yerine süreç sayısı kadar **MPI_Send** ve her süreçte buna mukabil bir **MPI_Recv** kullanılırsa sonuç yine aynı olur. Şekil 3.7.'de bu fonksiyonun sonuçları 4 süreç üzerinden resmedilmiştir.



Şekil 3.7. MPI_Scatter fonksiyonunun gösterimi

Aşağıdaki programda bu fonksiyonun kullanımı gösterilmiştir:

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char *argv[])
{
   int rank,boyut,i;
   double gonderme_tamponu[100],alma_tamponu;
   int gonderme_sayisi,alma_sayisi;
   int kok = 3;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&boyut);
   alma_sayisi=1;
   if(rank==kok)
   {
      for(i=0;i<boyut;++i)
      {
         gonderme_tamponu[i]=10.0+i;
      }
      gonderme_sayisi=1;
   }
   MPI_Scatter(gonderme_tamponu,gonderme_sayisi,MPI_DOUBLE,
      &alma_tamponu,alma_sayisi,MPI_DOUBLE,kok,MPI_COMM_WORLD);
   for(i=0;i<boyut;++i)
   {
      if(rank==i)
      {
```

```
            printf("P:%d alma_tamponu = %f \n",rank,alma_tamponu);
        }
        fflush(stdout);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Programda önce kök süreç tarafından gönderme tamponu oluşturulmaktadır. Daha sonra **MPI_Scatter** fonksiyonu ile gönderilen dizinin 0. elemanından başlanıp her süreç rankına göre ilgili adresten birer **MPI_DOUBLE** miktarı kadar veri okumaktadır.

Bu program dört süreç üzerinde çalıştırıldığı zaman her süreç çıktıya aldığı elemanı yazar ve bunların tek çıktıdaki sonucu şöyledir:

```
P:0 alma_tamponu = 10.00
P:1 alma_tamponu = 11.00
P:2 alma_tamponu = 12.00
P:3 alma_tamponu = 13.00
```

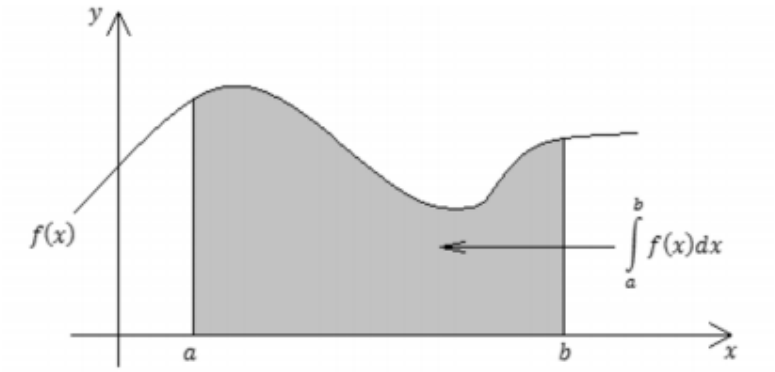
5. UYGULAMALAR

Tezin bu bölümünde bazı problemlerin paralel programlama ile çözülmesi gösterilmiştir. Bu problemler nümerik integral hesabı ve matris vektör çarpımı problemleridir.

İlk önce problemin kendisi açıklanmış daha sonra nasıl paralelleştirildiği gösterilip ilgili problem çözülmüştür.

5.2. Nümerik İntegral

$[a, b]$ aralığında negatif olmayan ve sürekli olan bir $f(x)$ fonksiyonunun belirli integrali x ekseni, $x = a$ ile $x = b$ düşey doğruları ve $f(x)$ fonksiyonunun bu aralıkta olan grafiğinin belirlediği alan olarak düşünülebilir. Şekil 5.1.'de bu durum gösterilmiştir (Pacheco, 1997).

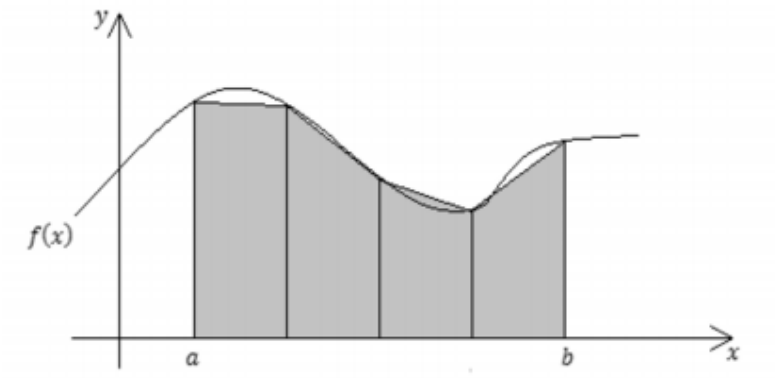


Şekil 5.1. Belirli integralin alan ile gösterilmesi

Böyle bir önkabulden sonra, bu alanı hesaplamak demek verili aralıkta ilgili fonksiyonun belirli integralini hesaplamaya denktir. Nümerik analizde, bu alanı hesaplamak için farklı yöntemler geliştirilmiştir. Bu yaklaşımların ortak özelliği verilen aralıkta alt sınırdan başlanıp eşit aralıklarla üst sınıra kadar düşey doğrular çekerek bölgeyi düzgün geometrik şekillere ayırıp, bu şekillerin alanlarının toplamını da bölgenin alanı gibi kabul etmektir. Düşey doğruların sayısı ne kadar fazla ise sözkonusu tahmin de gerçeğe o kadar yakındır. Yöntemleri belirleyen farklılıklar, iki düşey çizginin tanımladığı şeklin fonksiyonun grafiğine yakın olan tarafındaki kısmının nasıl tanımlandığında ortaya çıkar.

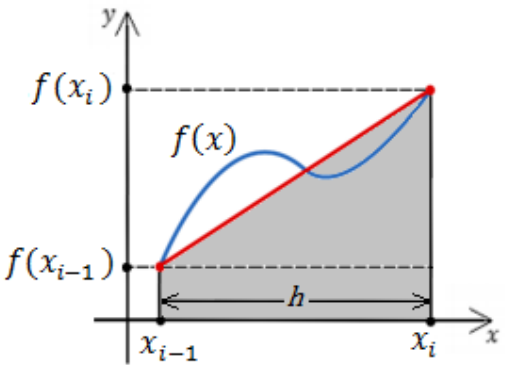
5.1.1. Yamuk yöntemi ile nümerik integral

Bu yöntemde düzgün geometrik şekiller yamuklardır. Bu durum Şekil 5.2.'de gösterilmiştir. Her yamuğun tabanı x ekseninde tanımlıdır. Düşey kenarlarının $f(x)$ fonksiyonunun grafiğini kestiği noktalardan geçen doğru parçası da yamuğun üst kenarını belirler.



Şekil 5.2. Alana yamuklar işe yaklaşmak

Tüm tabanların aynı uzunlukta olmalarını kabul etmek hesabı kolaylaştırır. Dolayısıyla n adet yamuk varsa her yamuğun tabanı $h = (b-a)/n$ olacaktır. En soldaki yamuğun tabanını $[a, a+h]$ aralığı temsil edecektir. Bir sonrakinin alanı $[a+h, a+2h]$ aralığı ile temsil edilcektir. Genel olarak, $i = 1,2,\dots,n$ olarak alınırsa i. yamuk $[a+(i-1)h, a+ih]$ aralığı ile temsil edilir. $x_i = a+ih, i = 0,\dots,n$ olarak kabul edilirse $i$nci yamuğun sol kenarının uzunluğu $f(x_{i-1})$ sağ kenarının uzunluğu da $f(x_i)$ olacaktır (Şekil 5.3.).



Şekil 5.3. İndisi *i* olan yamuğun gösterimi

Bu durumda yamuğun alanı

$$\frac{1}{2}h[f(x_{i-1}) + f(x_i)] \tag{5.1}$$

olur. Tüm yamukların alanlarının toplamı da

$$\frac{1}{2}h[f(x_0) + f(x_1)] + \frac{1}{2}h[f(x_1) + f(x_2)] + \cdots + \frac{1}{2}h[f(x_{n-1}) + f(x_n)]$$

olur ve bu eşitlikten

$$\left[\frac{f(x_0)}{2} + \frac{f(x_n)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1})\right]h \tag{5.2}$$

elde edilir.

$f(x)$ fonksiyonunu bir alt fonksiyon şeklinde yazdıktan sonra bu hesabı aşağıdaki fonksiyon ile seri bir şekilde gerçekleştirebilirz:

```
double YamukHesapla(double altSinir, double ustSinir,
                    int adimSayisi, double h)
{
   double integral;
   double x;
   int i;
   integral = (f(altSinir) + f(ustSinir))/2.0;
   x = altSinir;
   for (i = 1; i <= adimSayisi-1; i++)
   {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
}
```

Burada **altSinir** ve **ustSinir** sırasıyla a ve b değerlerini, **adimSayisi** n değerini temsil etmektedirler.

5.1.2. Simpson yöntemi ile nümerik integral

Yamuk yönteminden daha hassas bir yöntem olan Simpson yöntemi $f(x)$ fonksiyonunun grafiğine doğru parçaları ile değil de parabol yayları ile yaklaşmak fikrini baz alır. $p < q$ sayılarının reel olduğu varsayılırsa ve r sayısı $[p, q]$ aralığının

orta noktası olduğu kabul edilirse $h = (q-p)/2$ olması durumunda $(p, f(p))$, $(r, f(r))$ ve $(q, f(q))$ noktalarından geçen parabol

$$P(x) = \frac{f(p)}{2h^2}(x-r)(x-q) - \frac{f(r)}{h^2}(x-p)(x-q) - \frac{f(q)}{2h^2}(x-p)(x-r)$$

fonksiyonu ile hesaplanır. Bu fonksiyonun $\int_p^q P(x)dx$ biçiminde gösterilen integrali alınırsa

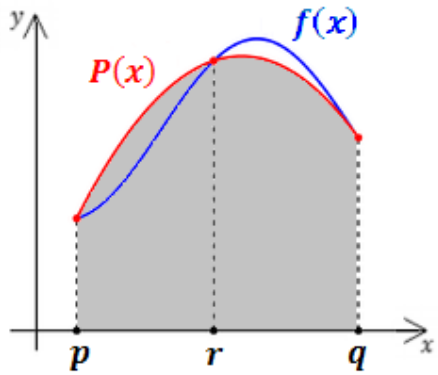
$$\frac{h}{3}[f(p) + 4f(r) + f(q)] \qquad (5.3.)$$

ifadesi elde edilir.

Yamuk yönteminde olduğu gibi $[a,b]$ aralığını n adet alt parçaya ayırırsak ve bu sayı çift olursa $\int_a^b f(x)dx$ integraline şu şekilde yaklaşabiliriz (Pacheco, 1997):

$$\int_a^b f(x)dx = \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)] \qquad (5.4)$$

Bu düzgün geometrik şekillerden biri Şekil Şekil 5.4.'te gösterilmiştir.



Şekil 5.4. Fonksiyona parabol ile yaklaşmak

$f(x)$ fonksiyonunu bir alt fonksiyon şeklinde yazdıktan (5.4)'te gösterilen hesabı aşağıdaki kod örneği ile seri bir şekilde gerçekleştirebilirz:

```
double SimpsonHesapla(double altSinir, double ustSinir,
                      int adimSayisi, double h)
{
   double integral;
   double x;
   int i;
   integral = (f(altSinir) + f(ustSinir))/2.0;
   x = altSinir;
```

```
    for (i = 1; i <= adimSayisi-1; i++)
    {
       x = x + h;
       if (i % 2 != 0)
       {
          integral = integral + 4 * f(x);
       }
       else
       {
          integral = integral + 2 * f(x);
       }
    }
    integral = integral*(h/3.0);
    return integral;
}
```

Burada **altSinir** ve **ustSinir** sırasıyla a ve b değerlerini, **adimSayisi** n değerini temsil etmektedirler.

5.1.3. Paralel nümerik integral

Yukarıda açıklanan yöntemleri paralel yöntemle çözmek için paralel programlama yaklaşımlarından biri olan veri paralelleştirme yaklaşımı kullanılmıştır. Bu yaklaşımın nümerik integrali uygulanabilmesinin matematiksel temeli belirli integralin kurallarından birine dayanmaktadır. Bu kurala göre $f(x)$ fonksiyonu $[a, b]$ aralığında sürekli ise ve $a < c < b$ için

$$\int_a^b f(x)dx - \int_a^c f(x)dx + \int_c^b f(x)dx$$

olur.

Bu yaklaşımda aralık altaralıklara bölünüp süreçlere atanır ve her süreç kendi veri kısmında aynı programı çalıştırır. MIMD sistemlerde tüm süreçlerde aynı programın çalışması yaklaşımı SPMD yaklaşımı olarak da bilinir.

Örneğimizdeki problemde alanı hesaplanacak bölge *n* adet alt şekile ayrılmıştır. *p* adet sürecin olduğunu düşünürsek ve *p*'nin *n*'yi kalansız böldüğünü varsayarsak her sürece *n/p* kadar bölge düşer. Hangi sürece hangi aralığın düştüğü Çizelge 5.1.'de gösterilmiştir.

Çizelge 5.1. Altaralıkların süreçlere atanması

Süreç	Aralık
0	$[a, a + \frac{n}{p}h]$
1	$[a+\frac{n}{p}h, a+2\frac{n}{p}h]$
⋮	⋮
i	$\left[a+i\frac{n}{p}h, a+(i+1)\frac{n}{p}h\right]$
⋮	⋮
$p-1$	$\left[a+(p-1)\frac{n}{p}h, b\right]$

Program önce MPI için gereken başlangıç işlemleri yaptıktan sonra **VerileriAl** fonksiyonuna girip burada sırasıyla alt sınır, üst sınır, adım sayısı ve yöntem bilgilerini kök sürecin (bu durumda 0. süreç) standart girişinden almaktadır. Daha sonra aldığı bu parametreleri süreç grubundaki diğer süreçlere **MPI_Send** ile göndermektedir. Kök dışındaki süreçler **MPI_Recv** ile bu değerleri almaktadırlar.

Daha sonra her sürecin hesap yapacağı alt aralıklar belirlenmektedir. İçerikleri sadece teker teker süreçlerde önem arzeden değişkenler "yerel" kelimesi ile başlamaktadırlar. Bu yaklaşım prorgamın dokümantasyonu için önemlidir (Pacheco, 1997).

Daha sonra **Hesapla** fonksiyonu çağrılmıştır. Bu fonksiyon seçilen yönteme göre yukarıda belirtilen hesaplardan birini yapıp sonucu ana programa intergral değişkenine döndürmektedir. Hatırlatılması gereken nokta bu fonksiyonun altaralıktaki integrali hesaplamasıdır. Bu noktadan sonra tüm süreçler **MPI_Barrier** fonksiyonu ile senkronize olduktan sonra **MPI_Reduce** fonksiyonu **MPI_SUM** işlemi ile kök süreç için çağrılıp alttoplamlar toplam değişkeninde toplanırlar.

En sonunda da kök sürecin standart çıkışına hesapın sonucu anlamlı bir şekilde yazılmaktadır.

Programın kaynak kodu EK I'deki 1. program olarak verilmiştir.

Paralel programlardaki giriş/çıkış konusu önemlidir. Nümerik integral programında tüm girişler ve çıkışlar kök süreç üzerinden yapılmaktadır. Eğer her süreç giriş/çıkış yapmaya kalkarsa bilhassa girişte istenmeyen sonuçlar ortaya çıkabilir. Farklı süreçler girişin farklı kısımlarını alıp hesapta karışıklıklar olabilir. Onun için en güvenli yol bir süreç üzerinden bu işi yapıp diğer tüm süreçlere ilgili verileri göndermektir (Pacheco, 1997).

5.2. Matris Vektör Çarpımı

Yönetici/işçi modelinde yönetici işçilerinin sayısından fazla işleri kendinde tutar ve işçiler müsait oldukça bu işleri işçilere atayıp sonucun gelmesini bekler. Sonuç geldikçe ilgili işçiye yeni bir iş atanır.

Bu uygulamada **a** kare matrisi **b** vektörü ile çarpılıp sonuç **c** vektöründe toplanmaktadır. Uygulama yönetici/işçi modeline göre yazılmıştır. 0. süreç yönetici süreç olarak belirlenmiştir. Yönetici kodu, işçilere matrisi satır satır göndermektedir. Kendilerine gelen satırı alan işçiler de **b** vektörü ile satırı nokta çarpımına tabi tutup sonucu yöneticiye göndermektedir (Gropp ve ark., 2003).

Bu uygulamanın kaynak kodu EK-1'de verilmiştir.

4. PARALEL BİLGİSAYAR KURULUMU

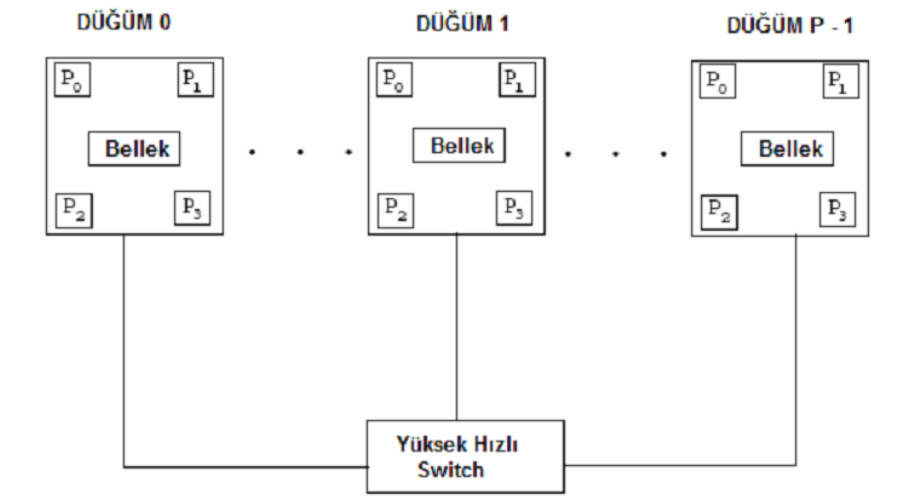
Yüksek lisans süresince çalışmalar iki yönde yapıldı. Birincisi paralel hesaplama, paralel algoritma geliştirme, paralel hesaplamanın farklı alanlardaki kullanımına dair teorik olarak nitelenebilen incelemer şeklinde gelişti. Çalışmaların bir diğer kısmını da paralel bilgisayarların nasıl kurulduğuna dair incelemeler ve pratikler oluşturdu (Kaçka ve ark., 2010). Halihazırdaki bir ortamı kullanabilmek ne kadar önemli ise de bir ortamı kurabilmek de önem taşımaktadır.

Bu bölümde paralel programların çalıştırılması için gereken HPC (*High Performance Computing* – Yüksek Başarımlı Hesaplama) ortamının kurulumu örnek iki bilgisayar üzerinden açıklanmıştır. Birden çok aşamadan oluşan kurulum işleminin her aşamasına imkan veren teknolojiler özetlenmiş ve kurulumdaki yansımaları açıklanmıştır. İki bilgisayar ile yapılan örnek kurulum, ikiden çok bilgisayar kullanılarak da aynı şekilde gerçekleştirilebilir. Genelde bilgisayar sayıları 2'nin kuvvetleri şeklinde seçilmektedir.

4.1. Paralel Bilgisayar Çeşidi

Paralel bilgisayar çeşidi olarak Beowulf Kümesi seçildi. Paralel hesaplama dünyasında popüler ve ucuz bir çözüm olarak bilinen Beowulf Kümeleri üzerinde Linux işletim sistemi çalışan ticari bilgisayarlardan oluşmaktadırlar. Düğümler olarak bilinen bu bilgisayarlar yüksek hızdaki ağlar ile birbirlerine bağlıdırlar. Aşağıdaki şekilde bir prototipi gösterilen Beowulf Kümeleri yüksek başarımlı hesaplama dünyasında hem bilimsel uygulamalarda hem de ticari uygulamalarda önemli bir yer tutmaktadırlar (Karniadakis ve ark., 2003).

İlk Beowulf Kümesi 1994 yılında NASA'da GFLOPS performansına ulaşmak amacı ile oluşturuldu. Standart bir Ethernet ağı ile 16 PC birbirine bağlandı. Her PC'de bir adet 486 Intel işlemcisi vardı ve yaklaşık olarak 70 MFLOPS performansında idi. Proje 40.000 Amerikan dolarına mal oldu ve o zamanda, aynı performansa sahip bir süperbilgisayarın fiyatı 1.000.000 Amerikan doları idi. Ortaçağda Grendel adlı devi yenen kahramanın anısına bu yapıya *Beowulf* adı verildi (Karniadakis ve ark., 2003).



Şekil 4.1. Bilgisayar kümesi örneği

4.2. Kurulumda Kullanılan Bilgisayarlar

Örnek kurulumda iki adet bilgisayar kullanılmıştır. Bunlar sırasıyla C0 ve C1 olarak isimlendirilmişlerdir. Sahip oldukları işlemci çeşitleri ve RAM kapasiteleri aşağıdaki çizelgede özetlenmiştir:

Çizelge 4.1. Örnek kurulumda kullanılan bilgisayarların işlemci çeşidi ve RAM kapasiteleri

Bilgisayar	İşlemci	RAM
C0	Intel® Core™ 2 Duo (2.9 GHz)	2 GB
C1	Intel® Pentium™ III (800 MHz)	512 MB

4.3. İşletim Sistemi

Beowulf Kümesinde bulunacak her bilgisayara Linux™ işletim sisteminin yaygın olarak kullanılan dağıtımlarından biri olan Fedora™ 14 kurulmuştur.

4.3.1. Linux işletim sistemi

Linux işletim sistemi daha önce geliştirilen Unix™ işletim sisteminin çekirdeğini kullanan, çıkarıldığı 1991 yılından sonra çok hızlı bir şekilde tüm dünyaya yayılan ve

tüm dünyadaki çok sayıda programcı tarafından geliştirilen, değiştirilen açık kaynak kodlu bir işletim sistemidir.

4.3.1.1. Linux işletim sisteminin özellikleri

Hangi dağıtım kullanılırsa kullanılsın tüm Linux'larda ortak olan kod kısmı çekirdek (*kernel*) olarak bilinir. Her Linux çekirdeği şu özellikleri sağlar (Negus, 2010):

- Çoklu kullanıcı özelliği (*multiuser*) – Birden çok kullanıcı hesabı tanımlayabilme özelliği yanında bir Linux sistemine birden çok kullanıcı girip aynı anda kendi işlerini yürütebilirler.
- Çoklu görev özelliği (*multitasking*) – Bir Linux sisteminde aynı anda birden çok program çalışabilir. Bunlar kullanıcıların işlerini gören uygulama yazılımları olabileceği gibi arkada çalışan sisteme has yazılımlar da olabilirler. *Daemon* denilen bu programlar sayesinde Linux işletim sistemi güçlü bir sunucu sistemi haline gelmektedir.
- Donanım desteği özelliği – Linux, bilgisayara bağlanabilecek herhangi bir donanımı destekleyen bir işletim sistemidir.
- Ağa bağlanabilirlik özelliği – Bir Linux sistemini bir ağa bağlamak için Linux çok sayıda ağ cihazını ve ağ protokolünü desteklemektedir. Bu ağ protokollerinden bazıları TCP/IP, IPX ve X.25 gibi protokollerdir.
- Ağ sunucusu özelliği – Bir yerel alan ağındaki (LAN – Local Area Network) veya İnternet'teki istemcilere servisler sağlamak Linux'un yaptığı en iyi işlerdendir. Linux'u bir FTP sunucusu, Web sunucusu, yazıcı sunucusu veya DHCP sunucusu olarak kullanmak çok kolay ve kullanışlıdır.

4.3.1.2. Linux dağıtımları

Bir Linux çekirdeğinin yanında gelen ve Linux'u daha kullanışlı ve yaygın bir hale getirmeye çalışan paket yazılımların tümü bir Linux dağıtımını (*distribution*) tanımlarlar. Bir Linux sistemindeki tüm arayüzler, uygulama yazılımları, derleyici, IDE gibi yazılım geliştirme yazılımları dağıtımın belirlediği programlardır. Bir Linux

dağıtımının esas amacı birbirleri ile ilişkili olmayan binlerce yazılım paketini bir bütün halinde bir araya getirmektir.

Linux çekirdeği gezegendeki en yaygın çekirdektir. Bu Linux'un çok sayıda farklı CPU ve cihazda çalıştığı anlamına gelmektedir. Bunların yanında bu cihazları destekleyen bir o kadar da dağıtım vardır. Bu dağıtımlar arasında en yaygın olanları Debian, Ubuntu, openSUSE, Slackware ve Fedora gibi dağıtımlardır.

4.3.1.3. Fedora dağıtımı

Uzun yıllar boyunca en popüler ticari dağıtımlardn bir Red Hat Linux dağıtımı idi. 2003 yılının Eylül ayından sonra Red Hat, Inc. şirketinin çalışma şeklini değiştirmesi sonucu bu dağıtım iki şekilde yoluna devam etti:

- Fedora Projesi (Fedora Project) – Bir açık kaynak projesi olan Red Hat Linux 9 baz alınarak kendi yolunda gelişen bir projedir. Proje kendi dağıtımlarını geliştirmektedir.
- Red Hat Kurumsal Linux (Red Hat Enterprise Linux) – Red Hat, Inc. şirketi tarafından sağlanan ticari Linux ürünlerinden oluşan resmi bir işletim sistemi. Yıllık üyelik şeklinde sunulan bu ürünler teknik destek, eğitim ve dokümantasyon ile kullanıcılara ulaşmaktadırlar.

Fedora Projesinin esas sonuçları Fedora olarak bilinen Linux dağıtımını belirleyen yazılım paketlerinin ortaya çıkmalarıdır (Negus, 2010).

4.3.2. Bir Beowulf Kümesinde Linux kullanmanın nedenleri

Bir Beowulf Kümesinde Linux kullanmanın en önemli sebeplerinden biri Linux işletim sisteminin esnek bir sistem olmasıdır (Gropp ve ark., 2003). Açık kaynak bir sistem olan Linux istenilen görevin ihtiyaçlarına göre kolaylıkla değiştirilip yeniden düzenlenebilir. Bir Beowulf Kümesinde kullanmadan önce her Linux sistemini değiştirmek gerekmez. Bazı ekstrem durumlarda, çok spesifik problemlerin dayattığı şartlar işletim sisteminde bazı modifikasyonları gerektirebilir. Nitekim burada bahsedilen kurulumda Fedora 14 dağıtımında herhangi bir modifikasyon yapılmamıştır.

Bir diğer neden de Linux kullanıcısının asla yalnız kalmayacağıdır. Linux kültürünün en önemli özelliklerinden biri ihtiyacı olan insanlara tecrübelilerin yardım etme çabalarıdır. Linux tecrübesi olan ve Linux ile çalışmaktan zevk alan bilgili sistem

yöneticilerinin sayısı çok fazladır ve bunlar bu bilgileri paylaşmaktan geri durmazlar. Daha özelde ise bu işletim sistemi ile Beowulf Kümesi kuranların sayısı ve bilgileri de azımsanmayacak seviyededir. Her geçen gün Linux kümelerinin sayısı artmaktadır. Bunlar ya bir kampüsün laboratuarında ya da bir şirketin yoğun işlemlerini karşılamak amacıyla şirketin sunucu merkezlerinde kurulmaktadırlar. Dünyada bilinen en güçlü 500 bilgisayar sistemini sıralayan ve detaylandıran TOP500 projesine (www.top500.org) bir göz atılırsa Linux işletim sisteminin yaygınlığı görülebilir. Google şirketi (www.google.com) bir bilimsel hesaplama kümesi olmamasına rağmen şirketin sunucularında Linux işletim sistemi çalışmaktadır ki bu durum da Linux işletim sisteminin esnekliğini ve uyumluluğunu göstermektedir (Gropp ve ark., 2003).

Linux'un tercih edilmesinin bir diğer nedeni de çok sayıda işlemciyi desteklemesidir. Bu da çok farklı mimariler kullanarak paralel hesaplama kümesi kurulabileceği anlamına gelmektedir. Mevcut işlemcileri desteklemesi yanında yeni çıkan işlemcilere de çok hızlı bir şekilde uyum sağlamaktadır.

Bu işletim sisteminin tercih edilmesinin en önemli sebeplerinden biri de çekirdeğinin ihtiyaçlara göre küçültülebilir olmasıdır. Bu kullanılmayan özelliklerin çekirdekten çıkartılması ve işletim sisteminin sadece özel işleri yerine getirmesi anlamına gelmektedir. "Daha küçük daha iyidir" ilkesinin nedenlerinden biri yıllarca süren kaynak kod geliştirme tecrübesinden gelmektedir. Kaynak ağacına bir satır kod eklendiği zaman gizli bir hatanın yerleştirilmesi olasılığı artar. Paralel hesaplama gibi işlem gücü yüksek olan ve uzun süreler alabilen hesaplamalarda istikrar önemlidir. Daha küçük bir çekirdek muhtemelen daha istikrarlı olacaktır.

Küçük çekirdeğin tercih edilmesinin bir diğer nedeni de en istikrarlı kod yolunun en çok kullanılan kod yolu olmasıdır. Çekirdek ne kadar küçükse test edilmeyen kod yollarının sayısı da o kadar azdır. Bilimsel hesaplamada bilimsel uygulama tarafından kullanılmayan her komut ve bilimsel problem çözümünün daha performanslı olması için kullanılması gerekirken ve fakat çekirdek tarafından gereksiz yere işgal edilen her önbellek baytı hesaplamayı gereksiz yere uzatacak, zora sokacaktır. Dolayısıyla Beowulf Kümelerinde küçük çekirdek tercih edilen bir durumdur (Gropp ve ark., 2003).

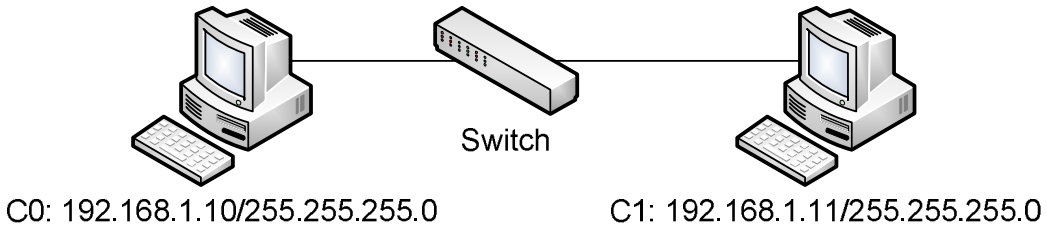
4.3.3. İşletim sistemini kurarken dikkat edilen hususlar

Her makinaya kurulan işletim sisteminde parolası aynı olan **cluster** adında birer kullanıcı oluşturulmuştur. Bunun nedeni MPI uyarlaması olan MPICH2'den gelmektedir. Sözkonusu uyarlama programları haberleştirirken SSH (Secure Shell) kullanmaktadır ve bu SSH her makinadan her makinaya şifresiz olmak zorundadır. Daha sonra gösterilecek bu ayarlama özel anahtar kriptografisini (*private-key cryptography*) kullanmaktadır ve aynı kullanıcı adı ile aynı parolaya sahip olmak bu kurulumu kolaylaştırmaktadır.

4.3.4. Ağ ayarı

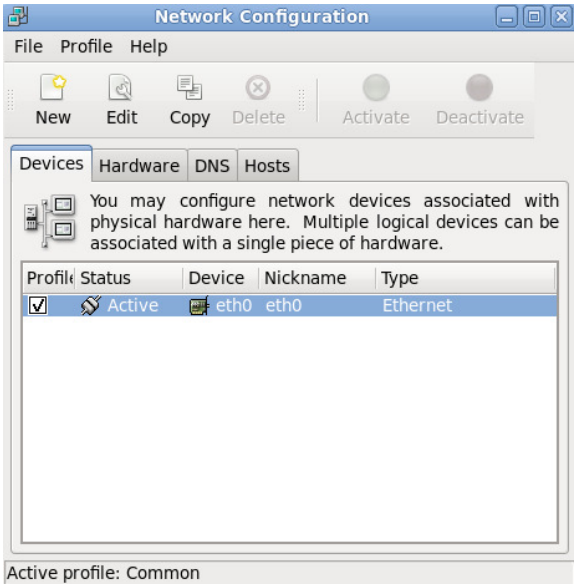
Beowulf Kümesindeki her bilgisayarın birbirleri ile haberleşmeleri gerekmektedir. Bunun için kurulumdaki bilgisayarları kapsayan bir yerel alan ağı kurulmuştur. Ağ türü olarak Yıldız Topolojisi seçilmiştir.

Bilgisayarlar için 192.168.1.0/255.255.255.0 ağı seçilmiş ve C0 bilgisayarına 192.168.1.10/255.255.255.0 ve C1 bilgisayarına da 192.168.1.11/255.255.255.0 statik IP adresleri atanmıştır. Şekil 4.2.'de kurulan ağ özetlenmiştir.



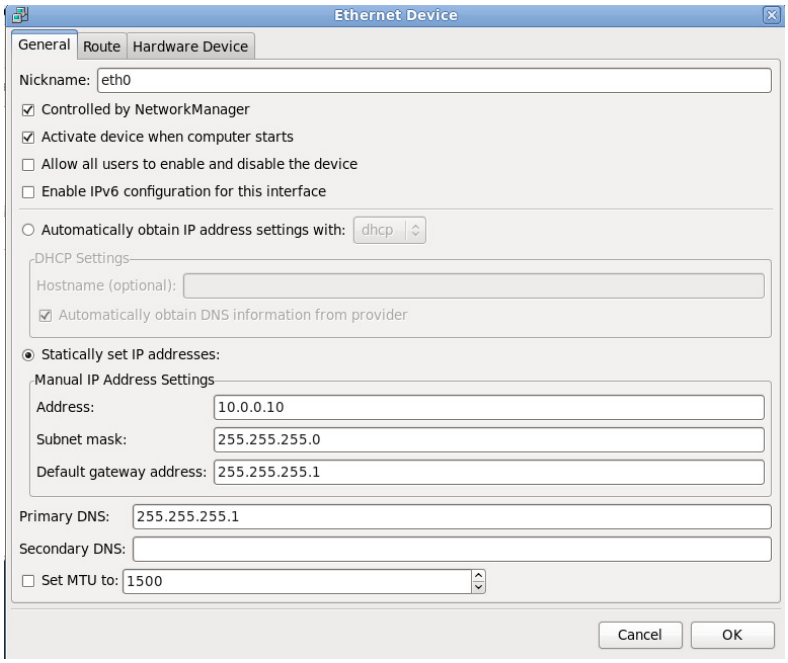
Şekil 4.2. Kurulan yerel alan ağı

Fedora 14 ortamında bu ağ ayarını gerçekleştirmek için C0 bilgisayarında **System** menüsünden **Administration** alt menüsünden **Network** seçeneği seçilip aşağıda gösterilen **Network Configuration** arayüzü açılmıştır:



Şekil 4.3. Ağ Ayarlama arayüzü

Daha sonra eht0 seçeneği çift tıklandıktan sonra ilgili cihaza aşağıdaki araüzde gösterildiği şekilde IP adres ayarı yapılmıştır:



Şekil 4.4. Ethernet cihazına IP adresi atama arayüzü

Aynı ayarlamalar C1 bilgisayarında da yapılmıştır.

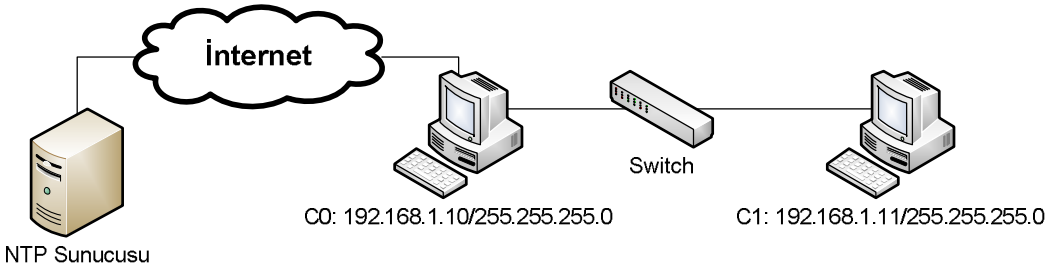
Daha sonra bilgisayarların birbirlerini isimlerinden tanımaları için **/etc/hosts** dosyaları düzenlenmiştir. Bu dosya bilgisayarın IP adreslerini isimleri ile eşleştirmek için kullanılır. C0 ve C1 bilgisayarlarının **/etc/hosts** dosyalarına

192.168.1.10 C0
192.168.1.11 C1

olmak üzere ikişer satır eklenmiştir. Daha sonra **ping** komutu ile her iki bilgisayara erişimin olup olmadığı kontrol edilmiştir.

4.3.4. NTP ayarı

NTP (*Network Time Protocol*) protokolü bir ağdaki bilgisayarların saatlerinin senkronizasyonu için kullanılır. Beowulf Kümesindeki tüm bilgisayarların saatlerinin birbirleri ile eşzamanlı olmaları gerekiyor. Bunun için sunucu olarak kabul edilen C0 bilgisayarı İnternet'teki bazı NTP sunucuları ile kendini senkronize edip, kümedeki diğer bilgisayarlar (örneğimizde sadece C1 bilgisayarı) da C0 bilgisayarını NTP sunucusu olarak gösterirler. Bu senaryoyu gerçekleştirmek için C0 bilgisayarının İnternet'e erişiyor olması gerekir. Bu ya ana makina kabul edilen bilgisayara ikinci bir Ethernet kartı takılarak yapılır, ya da mevcut ethernet kartına ikinci bir IP adresi atanır (bu atamanın İnternet'e çıkması gerekir) ya da modem üzerinde bazı ayarlamalar yapılıp ana bilgisayarın İnternet'e çıkması sağlanır. NTP sunucusuna erişen ağımızın şeması aşağıdaki şekilde gösterilmiştir:



Şekil 4.5. NTP sunucusuna erişen yerel alan ağı

NTP ayarı yapmak için **/etc/ntp.conf** dosyasının düzenlenmesi gerekmektedir. Örnek kurulumda C0 makinasının ilgili dosyasına

server 0.fedora.pool.ntp.org iburst
server 1.fedora.pool.ntp.org iburst
server 2.fedora.pool.ntp.org iburst
server 3.fedora.pool.ntp.org iburst

ve C1 makanisanın ilgili dosyasına da

server C1

satırları eklenmiştir. Bu işlemden sonra önce C0 makinasında daha sonra C1 makinasında **root** kullanıcı hakkı ile şu komutlar çalıştırılmıştır:

 # /sbin/service ntpd restart

ntpd programı arkada çalışıp NTP protokolü ile ilgili işleri yürütmektedir. Bu işlemden sonra her iki bilgisayarın saati mikro saniye hata toleransında aynı olmuştur.

4.3.6. SSH ayarı

SSH (*Secure Shell*) paketi ağdaki diğer bilgisayara kabuk üzerinden erişip iş yaptırma imkanı sağlar. Bunun yaparken de ağ trafiğin şifreler. Bu paket özel anahtar kriptografisini kullanır, dolayısıyla iletişimin güvenliği sağlanmış olur. Bu paket üç komut sağlar. Bunlar

- **ssh** – uzaktan giriş/çalıştırma,
- **scp** – uzaktan kopyalama ve
- **sftp** – FTP (*File Transfer Protocol*) sitelerine güvenli erişim

komutlarıdır.

MPICH2 kurulum dokümanında anlatıldığı gibi her makinadan her makinaya parolasız **ssh** komutunun çalışması gerekmektedir. Bundan dolayı örnek kurulumda bu ayarlamalar yapılmıştır. Burada C0'dan C1'e olan ayarlamalar gösterilmiştir (Negus, 2010).

C0 makinasına **cluster** kullanıcısı ile giriş yaptıktan sonra **ssh** anahtarını üretmek için

```
$ ssh-keygen –t dsa
```

yazılıp üç defa <Enter> tuşuna basıldığında bunun sonucunda **/home/cluster/.ssh/id_dsa** dosyasına açık/gizli DSA (*Digital Signature Algorithm*) anahtar çifti üretilmiştir. Üretilen kimlik doğrulama (*authentication*) anahtarlarını güvenlikli hale getirmek için **$HOME** klasörüne, **.ssh** klasörüne ve kimlik doğrulama dosyalarına aşağıdaki komutlar ile hak ayarlamaları yapılmıştır:

```
$ chmod go-w $HOME
$ chmod 700 $HOME/.ssh
$ chmod go-rwx $HOME/.ssh/*
```

Daha sonra aşağıdaki komut ile açık anahtar karşı bilgisayara (C1) kopyalanmıştır. Bu işlem için C1 bilgisayarının **cluster** kullanıcısının parolasının girilmesi gerekir:

```
$ scp /home/cluster/.ssh/id_dsa.pub cluster@C1:/tmp
```

Aşağıdaki komut ile uzaktaki kullanıcıların yetkilendirme (*authorization*) anahtarlarına ssh anahtarı eklenmiştir:

```
$ ssh cluster@C1 'cat /tmp/id_dsa.pub >> /home/cluster/.ssh/authorized_keys2'
```

Karşı makinadaki **sshd** sürecinin *authorized_keys2* dosyasını kabul etmesi için $HOME klasörüne ve kimlik doğrulama dosyalarına güvenli erişimin sağlanması gerekir. Aşağıdaki komutlar bu hakları sağlamıştır:

```
$ ssh cluster@C1 chmod go-w $HOME
$ ssh cluster@C1 chmod 700 $HOME/.ssh
$ ssh cluster@C1 ''chmod go-rwx $HOME/.ssh/*''
```

Son olarak

```
$ ssh cluster@C1 rm /tmp/id_dsa.pub
```

komutu ile daha önce kopyalanan dosya silinmiştir.

Bu işlemlerden sonra C0'dan C1'e parolasız erişim sağlanmıştır. Daha sonra aynı işlemler C1'den C0'a, C0'dan C0'a ve C1'den C1'e yapılmıştır.

Yapılanları özetlemek gerekirse esas işlemler ilgili klasörlere hak verdikten sonra her makinanın üretilen açık anahtarını erişmek istediği makinanın **/home/cluster/.ssh/authorized_keys2** dosyasına kopyalamaktan ibarettir. Böylece bu makina diğerine parola kullanmadan erişebilecektir.

4.4. MPICH2 Kurulumu

MPICH2, MPI standartının (MPI-1 ve MPI-2) yüksek performanslı ve birden çok platformu destekleyen bir uyarlamasıdır. MPICH2'nin iki amacı vardır:

1. Farklı yapıdaki hesaplama ve iletişim platformlarını verimli bir şekilde destekleyecek bir MPI uyarlaması sağlamak.
2. Diğer türetilmiş uyarlamalar için sağlanan genişletilmesi kolay modüler bir altyapı aracılığıyla MPI alanında ileri araştırmalara imkan vermek.

MPICH2 Amerika Birleşik Devletleri'ndeki Argonne Ulusal Laboratuarı'nda geliştirilmektedir.

Kurulum için gereken paket olan **mpich2-1.3.1.tar.gz** dosyası C0 bilgisayarının **/home/cluster/libraries** klasörüne indirildikten sonra bu klasöre geçip

```
$ tar xfz mpich2-1.3.1.tar.gz
```

komutu ile dosya açılmıştır. Açılan klasörde MPICH2 kaynak kodu bulunmaktadır ve bunun derlenmesi gerekmektedir. Daha sonra

```
$ mkdir /home/cluster/mpich2-install
```

komutu ile bir kurulum klasörü oluşturulmuştur. Derleme klasörünü oluştumak için şu komut çalıştırılmıştır:

```
$ mkdir /tmp/cluster/mpich2-1.3.1
```

Daha sonra bu klasöre geçip kaynak klasördeki **configure** skripti çalıştırılıp kurulum klasörü belirtilerek aşağıdaki komut ile MPICH2 yapılandırılmıştır:

```
$ /home/cluster/libraries/mpich2-1.3.1/configure -prefix=/home/cluster/mpich2-install \
--with-device=ch3:sock --with-pm=mpd 2>&1 | tee c.txt
```

Burada **--with-device:ch3:sock** yapılandırma seçeneği prosesler arasındaki tüm iletişimler için soketlerin kullanılacağını belirtmektedir. **--with-pm:mpd** seçeneği ise MPICH2'deki geleneksel **mpd** (*multi-purpose daemon*) proses yöneticisinin kullanılacağını belirtmektedir. Paralel hesaba katılan her bilgisayarın arkaplanında bu program çalışmaktadır. Bu arkaplan programları bir halka oluşturup proseslerin başlatılmasını kolaylaştırırlar.

```
$ make 2>&1 | tee m.txt
```

komutu ile MPICH2 derlendikten sonra

```
$ make install 2>&1 | tee mi.txt
```

komutu ile kurulum oluşturulumuştur.

Bu adımlar sonucunda bir hata olup olmadığını kontrol etmek için sırasıyla oluşturulan **c.txt**, **m.txt** ve **mi.txt** dosyalarına bakılabilir.

```
$ scp -r /home/cluster/mpich2-install/* C1:/home/cluster/mpich2-install/
$ scp -r /tmp/cluster/mpich2-1.3.1/examples/* C1:/tmp/cluster/mpich2-1.3.1/examples/
```

komutları ile sırasıyla önce kurulum klasörü daha sonra paketten gelen örnekler klasörü aynı klasör yapısı şeklinde C1 makinasına kopyalanmıştır. Alternatif olarak NFS (*Network Files System*) gibi mekanizmalar kullanılarak bu klasörler kümedeki diğer makinalara paylaştırılabilirler.

```
$ sudo vi $HOME/.bashrc
```

komutu her iki makinada çalıştırılıp (daha doğrusu **vi** düzenleme aracı **$HOME/.bashrc** dosyasını düzenlemek amacı ile **root** kullanıcı hakları ile çalıştırılmıştır) açılan dosyaya

```
PATH=$PATH:/home/cluster/mpich2-install/bin; export PATH
```

satırı eklenmiştir. Bu sayede MPICH2 kurulum klasörüne her yerden erişim sağlanmıştır.

Son olarak her iki makinanın **$HOME** klasörlerine

```
$ touch .mpd.conf
$ chmod 600 .mpd.conf
```

komutları ile .mpd.conf dosyası oluşturulmuş ve bu dosyalara bir editör ile **MPD_SECRETWORD=mr45-j9z** satırı eklenmiştir. Kümedeki mpd arkaplan programları bu değeri kullanarak bir mpd halkasına girerler. Bunun her makinada ve aynı olması gerekmektedir.

Bundan sonra C0 bilgisayarında **$HOME** klasörüne **.mpd.hosts** adında bir dosya oluşturulmuş ve bu dosyaya **C1** değeri yazılmıştır. Bir mpd halkası C0'dan başlatılacağı için ilgili komut bu dosyadaki makinaların halkaya katılacağını bilmektedir.

Kurulumu test etmek için aşağıdaki komutlar çalıştırılmış ve aşaıdaki sonuçlar elde edilmiştir (çalışan komutların çıktıları eğik gösterilmiştir):

```
$ mpdboot -n 2
$ mpdtrace -l
C0_49439 (192.168.1.10)
C1_38005 (192.168.1.11)
$ mpiexec -l -n 2 /tmp/cluster/mpich2-1.3.1/examples/cpi
0: Process 0 of 2 is on C0
1: Process 1 of 2 is on C1
0: pi is approximately 3.1415926544231243, Error is 0.0000000008333312
0: wall clock time = 0.017561
$ mpdallexit
```

Burada **mpdboot** komutu ile iki makinada mpd programları başlatılmıştır. **mpdtrace** komutu ile çalışan mpd programları listelenmiştir. **mpiexec** komutu ile de paketle gelen örnekler klasöründeki π sayısını yaklaşık hesaplayan program çalıştırılmıştır. **mpdallexit** komutu ile de halkadaki tüm mpd programları sonlandırılmıştır.

5. SONUÇLAR VE ÖNERİLER

Bu tez çalışması sonucunda Beowulf kümelerinin nasıl kurulacağı konusunda önemli tecrübeler elde edilmiştir. Basit matematiksel problemlerin nasıl çözüleceği incelenmiştir. Diferansiyel denklemlerin çözümünde de kullanılan paralel hesaplama yöntemlerinin incelenmesi gerekmektedir.

KAYNAKLAR

Anonim, 2011, Introduction to MPI [online], Cyberinfrastructure Tutor, National Center for Supercomputing Applications at the University of Illinois, http://www.citutor.org [Ziyaret Tarihi: 21 Şubat 2011].

Barney B., 2010, Introduction to Parallel Computing [online], Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/ [Ziyaret Tarihi: 21 Şubat 2011].

Barney B., 2010, Message Passing Interface [online], Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/mpi/ [Ziyaret Tarihi: 21 Şubat 2011].

Grama A., Gupta A., Karypis G., Kumar V., 2003, Introduction to Parallel Computing, Second Edition, Addison Wesley.

Gropp W., Lusk E., Sterling T., 2003, Beowulf Cluster Computing with Linux, Second Edition, The MIT Press.

Heath M.T., 1997, Scientific Computing: An Introductory Survey, The McGraw-Hill Company.

Kaçka S., Dündar Ö., Keskin Y., Oturanç G., 2010, Setting Up a High Performance Computing Envionment in the Ereğli Kemal Akman Vocational School, *Selçuk Journal of Applied Mathematics Special Issue*, 3-18.

Karniadakis E. M., Kirby R. M., 2003, Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation, Cambridge University Press.

Negus C., 2010, Fedora® Bible 2010 Edition: Featuring Fedora Linux®, Wiley Publishing Inc.

Pacheco P. S., 1997, Parallel Programming with MPI, Morgan Kaufmann Publishers, Inc.

Parhami B., 2002, Introduction to Parallel Processing, Kluwe Academic Publishers.

Beowulf Project, http://www.beowulf.org/overview/projects.html, [Ziyaret Tarihi: 16 Haziran 2010]

Fedora Project, http://fedoraproject.org/, [Ziyaret Tarihi: 15 Ekim 2010]

MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/, [Ziyaret Tarihi: 16 Ekim 2010]

EKLER

EK-1 Örnek Uygulamalar Kaynak Kodları

```
//*****************************************************
//*** Nümerik İntegral Kaynak Kodu **********************
//*****************************************************
#include <iostream>
#include <string>
#include <mpi.h>

using namespace std;

// Sabit Tanımlamaları
#define KOK_SUREC 0
#define ALT_SINIR_ETIKETI 0
#define UST_SINIR_ETIKETI 1
#define ADIM_SAYISI_ETIKETI 2
#define YONTEM_ETIKETI 3
#define YAMUK_YONTEMI 1
#define SIMPSON_YONTEMI 2

// Fonksiyon Bildirimleri
void VerileriAl(double* altSinir, double* ustSinir, int* adimSayisi,
                                    int* yontem, int rank, int surecSayisi);
double Hesapla(double yerelAltSinir, double yerelUstSinir,
                            int yerelAdimSayisi, int yontem, double h);
double f(double x);
string YontemBelirle(int yontem);

// Programın Başlangıcı
void main(int argc, char** argv) {
   int              rank;
   int              surecSayisi;
   double           altSinir;
   double           ustSinir;
   int              adimSayisi;
   int              yontem;
   double           h;
   double           yerelAltSinir;          // alt aralığın alt sınırı
   double           yerelUstSinir;          // alt aralığın üst sınırı
   int              yerelAdimSayisi;    /   / alt aralıktaki adım sayısı
   double           integral;               // alt aralıktahesaplanan integral
   double           toplam;                 // toplam integral
   int              hedef = 0;
   MPI_Status       durum;

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

   // Verileri al ve süreçlere dağıt
   VerileriAl(&altSinir, &ustSinir, &adimSayisi, &yontem, rank, surecSayisi);

   h = (ustSinir-altSinir)/adimSayisi;
   yerelAdimSayisi = adimSayisi/surecSayisi;

   yerelAltSinir = altSinir + rank*yerelAdimSayisi*h;
```

```
   yerelUstSinir = altSinir + yerelAdimSayisi*h;

   // Yönteme göre integral hesapla
   integral = Hesapla(yerelAltSinir, yerelUstSinir, yerelAdimSayisi, yontem, h);

   // Süreçleri senkronize et
   MPI_Barrier(MPI_COMM_WORLD);
   // Tüm süreçlerdeki integral ile temsil edilen sonuçları toplam değişkenine
   //  Toplam işlemi ile indirge (toplu iletişim fonksiyonu)
   MPI_Reduce(&integral, &toplam, 1, MPI_DOUBLE, MPI_SUM,
                     KOK_SUREC, MPI_COMM_WORLD);

   // Sonucu standart çıkışa yaz
   if (rank == KOK_SUREC)
   {
      cout << adimSayisi << " adım sayısı ve " <<
                              YontemBelirle(yontem) << " ile " << endl;
      cout << "[" << altSinir << "," << ustSinir
                              << "] aralığında fonskyionun" << endl;
      cout << "integrali = " << toplam << endl;
   }
   MPI_Finalize();
}

// Yöntem belirleyen fonksiyon
//  gelen yontem parametresini iki
//  yöntemden birini temsilen bir string
//  değer döndürür
string YontemBelirle(int yontem)
{
   string sonuc = "";
   if (yontem == YAMUK_YONTEMI)
   {
      sonuc = "Yamuk Yöntemi";
   }
   else if (yontem == SIMPSON_YONTEMI)
   {
      sonuc = "Simpson Yöntemi";
   }
   return sonuc;
}

// Verileri alıp gruptaki süreçlere gönderen fonksiyon
void VerileriAl(double* altSinir, double* ustSinir, int* adimSayisi,
                                          int* yontem, int rank, int p)
{
   int hedef;
   MPI_Status durum;

   if (rank == KOK_SUREC)
   {
      cout << "Alt sınır:" << endl;
      cin >> *altSinir;

      cout << "Üst Sınır:" << endl;
      cin >> *ustSinir;

      cout << "Adım Sayısı:" << endl;
      cin >> *adimSayisi;
```

```
      cout << "Yöntem (1: Yamuk, 2: Simpson):" << endl;
      cin >> *yontem;

      // Kök hariç diğer süreçlere değerleri gönder
      for (hedef = 1; hedef < p; hedef++)
      {
         MPI_Send(altSinir, 1, MPI_DOUBLE, hedef, ALT_SINIR_ETIKETI,
                                    MPI_COMM_WORLD);
         MPI_Send(ustSinir, 1, MPI_DOUBLE, hedef, UST_SINIR_ETIKETI,
                                    MPI_COMM_WORLD);
         MPI_Send(adimSayisi, 1, MPI_INT, hedef, ADIM_SAYISI_ETIKETI,
                                    MPI_COMM_WORLD);
         MPI_Send(yontem, 1, MPI_INT, hedef, YONTEM_ETIKETI,
                                    MPI_COMM_WORLD);
      }
   }
   else
   {
      // Kök süreçten gelen değerleri al
      MPI_Recv(altSinir, 1, MPI_DOUBLE, KOK_SUREC, ALT_SINIR_ETIKETI,
                              MPI_COMM_WORLD, &durum);
      MPI_Recv(ustSinir, 1, MPI_DOUBLE, KOK_SUREC, UST_SINIR_ETIKETI,
                              MPI_COMM_WORLD, &durum);
      MPI_Recv(adimSayisi, 1, MPI_INT, KOK_SUREC, ADIM_SAYISI_ETIKETI,
                              MPI_COMM_WORLD, &durum);
      MPI_Recv(yontem, 1, MPI_INT, KOK_SUREC, YONTEM_ETIKETI,
                              MPI_COMM_WORLD, &durum);
   }
}


// Her süreçte çalışan ve belirlenen yönteme göre
//  alt aralıktaki integrali bulan fonksiyon
double Hesapla(double yerelAltSinir, double yerelUstSinir,
                                 int yerelAdimSayisi, int yontem, double h)
{
   double integral;
   double x;
   int i;

   if (yontem == YAMUK_YONTEMI)
   {
      // Yamuk yöntemine göre hesap yap
      integral = (f(yerelAltSinir) + f(yerelUstSinir))/2.0;
      x = yerelAltSinir;
      for (i = 1; i <= yerelAdimSayisi-1; i++)
      {
         x = x + h;
         integral = integral + f(x);
      }
      integral = integral*h;
   }
   else if (yontem == SIMPSON_YONTEMI)
   {
      // Simpson yöntemine göre hesap yap
      integral = f(yerelAltSinir) + f(yerelUstSinir);
      x = yerelAltSinir;
      for (i = 1; i <= yerelAdimSayisi-1; i++)
      {
         x = x + h;
```

```
            if (i % 2 != 0)
            {
               integral = integral + 4 * f(x);
            }
            else
            {
               integral = integral + 2 * f(x);
            }
         }
         integral = integral*(h/3.0);
      }
      return integral;
}


// İntegral fonksiyonu
//  şu anda kare fonksiyonu
double f(double x)
{
   double donusDegeri;
   donusDegeri = x*x;
   return donusDegeri;
}

//*******************************************************
//*** Nümerik İntegral Kaynak Kodu Sonu ******************
//*******************************************************



//*******************************************************
//*** Matris Vektör Çarpımı Kaynak Kodu ******************
//*******************************************************
#include <iostream>
#include <mpi.h>

#define BOYUT 4
#define MIN( x, y ) ((x) < (y) ? x : y)

using namespace std;
void yonetici( int surecSayisi )
{
   double a[BOYUT][BOYUT], c[BOYUT];
   int i, j, gonderen, satir, gonderimAdedi = 0;
   double noktaCarpimi;
   MPI_Status durum;

   // matrisin rasgele oluşturulması,
   //  bir dosyadan okunabilir
   for (i = 0; i < BOYUT; i++ )
      for ( j = 0; j < BOYUT; j++ )
         a[i][j] = ( double ) i+j;

   for ( i = 1; i < MIN( surecSayisi, BOYUT ); i++ ) {
      MPI_Send( a[i-1], BOYUT, MPI_DOUBLE, i, i, MPI_COMM_WORLD );
      gonderimAdedi++;
   }
   // işçilerden gelen nokta çarpımları al
   for ( i = 0; i < BOYUT; i++ ) {
      MPI_Recv( &noktaCarpimi, 1, MPI_DOUBLE,
```

```
            MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &durum );
      gonderen = durum.MPI_SOURCE;
      satir    = durum.MPI_TAG - 1;
      c[satir] = noktaCarpimi;
      // eğer kaldiysa bu işçiye bir diğer satırı gönder
      if ( gonderimAdedi < BOYUT ) {
         MPI_Send( a[gonderimAdedi], BOYUT, MPI_DOUBLE, gonderen,
                gonderimAdedi + 1, MPI_COMM_WORLD );
         gonderimAdedi++;
      }
      else
      {
         // yapacak iş kalmadı, sonlandır
         MPI_Send( MPI_BOTTOM, 0, MPI_DOUBLE, gonderen, 0,
                MPI_COMM_WORLD );
      }
   }

   for (int ii = 0; ii < BOYUT; ii++)
   {
      for (int jj = 0; jj < BOYUT; jj++)
      {
         cout << a[ii][jj] << " ";
      }
      cout << endl;
   }
   cout << endl << endl;

   for (int cc = 0; cc < BOYUT; cc++)
   {
      cout << c[cc] << endl;
   }
}


void isci( int myrank )
{
   double b[BOYUT], c[BOYUT];
   int i, satir;
   double noktaCarpimi;
   MPI_Status durum;

   // b vekötürunu rasgele oluştur
   //  bir dosyadan okunabilir
   for ( i = 0; i < BOYUT; i++ )
      b[i] = i + 1.0;

   for (int ii = 0; ii < BOYUT; ii++)
   {
      cout << b[ii] << endl;
   }


   if ( myrank <= BOYUT ) {
      MPI_Recv( c, BOYUT, MPI_DOUBLE, 0, MPI_ANY_TAG,
              MPI_COMM_WORLD, &durum );
      while ( durum.MPI_TAG > 0 ) {
         satir = durum.MPI_TAG - 1;
         noktaCarpimi = 0.0;
         for ( i = 0; i < BOYUT; i++ )
```

```
            noktaCarpimi += c[i] * b[i];
         MPI_Send( &noktaCarpimi, 1, MPI_DOUBLE, 0, satir + 1,
                  MPI_COMM_WORLD );
         MPI_Recv( c, BOYUT, MPI_DOUBLE, 0, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &durum );
      }
   }
}



void main(int argc, char** argv) {
   int                  rank;
   int                  surecSayisi;

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


   if (rank == 0)
   {
      yonetici(surecSayisi);
   }
   else
   {
      isci(rank);
   }

   MPI_Finalize();
}
//*********************************************************
//*** Matris Vektör Çarpımı Kaynak Kodu Sonu *************
//*********************************************************
```

ÖZGEÇMİŞ

KİŞİSEL BİLGİLER

Adı Soyadı : Serdar KAÇKA
Uyruğu : Kosova
Doğum Yeri ve Tarihi : Prizren, 1981
Telefon : 05339698172
Faks :
e-mail : serdarkacka@gmail.com

EĞİTİM

Derece		Adı, İlçe, İl	Bitirme Yılı
Lise	:	**Gjon Buzuku Lisesi, Fen-Matematik Bölümü, Prizren, Kosova**	**2000**
Üniversite	:	**Selçuk Üniversitesi, Mühendislik Mimarlık Fakültesi, Bilgisayar Mühendisliği Bölümü, Selçuklu, Konya**	**2005**
Yüksek Lisans	:		
Doktora	:		

İŞ DENEYİMLERİ

Yıl	Kurum	Görevi
2006 - Devam Ediyor	**İşlem Coğrafi Bilgi Sistemleri Eğitim ve Mühendislik Ltd., Şti.**	**Yazılım Proje Koordinatörü**

UZMANLIK ALANI

Paralel Hesaplama, Paralel Bilgisayarlar, Coğrafi Bilgi Sistemleri, Coğrafi Veritabanı Tasarımı, Yazılım Mimarisi, Nesneye Yönelik Analiz ve Tasarım

YABANCI DİLLER

İngilizce, Sırpça, Arnavutça

YAYINLAR

Kaçka S., Dündar Ö., Keskin Y., Oturanç G.,2010, Setting Up a High Performance Computing Envionment in the Ereğli Kemal Akman Vocational School, *Selçuk Journal of Applied Mathematics Special Issue*, 3-18