MPI İle Paralel Programlama

Tahsin Gökalp Şaan

YÜKSEK LİSANS TEZİ

Matematik - Bilgisayar Anabilim Dalı

Ekim 2017

Parallel Programming With MPI

Tahsin Gökalp Şaan

MASTER OF SCIENCE THESIS

Mathematics - Computer Department

October 2017

MPI İle Paralel Programlama

Tahsin Gökalp Şaan

Eskişehir Osmangazi Üniversitesi
Fen Bilimleri Enstitüsü
Lisansüstü Yönetmeliği Uyarınca
Matematik - Bilgisayar Anabilim Dalı
Bilgisayar Bilimleri Bilim Dalı
YÜKSEK LİSANS TEZİ
Olarak Hazırlanmıştır

Danışman: Yrd. Doç. Dr. Ahmet Faruk Aslan

Ekim 2017

ONAY

Matematik - Bilgisayar Anabilim Dalı Yüksek Lisans öğrencisi Tahsin Gökalp Şaan'ın YÜKSEK LİSANS tezi olarak hazırladığı "**MPI İle Paralel Programlama**" başlıklı bu çalışma, jürimizce lisansüstü yönetmeliğin ilgili maddeleri uyarınca değerlendirilerek oy birliği ile kabul edilmiştir.

Danışman : Yrd. Doç. Dr. Ahmet Faruk Aslan

Yüksek Lisans Tez Savunma Jürisi:

Üye : Yrd. Doç. Dr. Ahmet Faruk Aslan

Üye : Doç. Dr. Alper Odabaş

Üye : Yrd. Doç. Dr. Mustafa Saltan

Fen Bilimleri Enstitüsü Yönetim Kurulu'nun ..... tarih ve ..... sayılı kararıyla onaylanmıştır.

Prof.Dr. Hürriyet ERŞAHAN
Enstitü Müdürü

ETİK BEYAN

Eskişehir Osmangazi Üniversitesi Fen Bilimleri Enstitüsü tez yazım kılavuzuna göre, Yrd. Doç. Dr. Ahmet Faruk Aslan danışmanlığında hazırlamış olduğum "**MPI İle Paralel Programlama**" başlıklı tezimin özgün bir çalışma olduğunu; tez çalışmamın tüm aşamalarında bilimsel etik ilke ve kurallara uygun davrandığımı; tezimde verdiğim bilgileri, verileri akademik ve bilimsel etik ilke ve kurallara uygun olarak elde ettiğimi; tez çalışmamda yararlandığım eserlerin tümüne atıf yaptığımı ve kaynak gösterdiğimi ve bilgi, belge ve sonuçları bilimsel etik ilke ve kurallara göre sunduğumu beyan ederim. 16/10/2017

Tahsin Gökalp Şaan

ÖZET

Paralel programlama bir işi yapabilecek birden çok birimin aynı zamanda bir problem üzerinde çalışabilecek şekilde programlanması işlemidir. Paralel programlama genellikle bir bilgisayarın çözmekte zorlanacağı işlemleri birden çok bilgisayara veya işlemciye paylaştırarak çözmeyi sağlar. Bundan dolayı finans modellemeleri, matematik, meteoroloji,... gibi geniş bir kullanım alanına sahiptir. Bu kullanım alanlarında paralel programlama teorik olarak işlemleri hızlandırdığı için zamandan tasarruf sağlar ve zor problemlerin çözümünü mümkün hale getirir. MPI ise paralel programlarla alakalı bir bilgisayar iletişim protokolüdür. Bu tez paralel programlama hakkında temel kavramlar, literatür taraması, MPI kütüphaneleri ve bu kütüphaneler yardımıyla örnek uygulamalar oluşturulmasını içermektedir. Ayrıca bir cluster oluşturularak yazılan örnek uygulamaların bu cluster üzerinden çalıştırılması anlatılmaktadır.

Anahtar Kelimeler : Paralel Programlama, MPI, MPICH

SUMMARY

Parallel programming is the process of programming multiple processors to work on a mutual problem. Parallel programming assists us by solving a problem with multiple computers and processors when a single computer can't handle. Therefore parallel programming used at a wide range such as finance modelling, mathematics, meteorology,… Parallel programming speeds up the process theoretically therefore saves time and solves difficult problems in these areas. MPI is a communication protocol related to parallel programs. This thesis includes; basic concepts of parallel programming, literature review, MPI libraries and writing sample applications with these libraries. Lastly, it's explained how these sample applications works on the created cluster.

Keywords : Parallel Programming, MPI, MPICH

TEŞEKKÜR

Bu çalışma boyunca yardımlarını esirgemeyen hocalarım **Yrd. Doç. Dr. Ahmet Faruk ASLAN** ve **Doç. Dr. Alper ODABAŞ**'a, beni destekleyen merhum babam **Hüseyin Şaan**'a, annem **Feryal Şaan**'a, eşim **Sezin Cirdi Şaan**'a, kardeşim **Tayfun Şaan**'a, çalışmanın şekil çizimlerinde büyük yardımları dokunan **Murat Çayır**'a sonsuz saygı ve teşekkürlerimi sunarım.

İÇİNDEKİLER

Sayfa

ÖZET . . . . . **vi**

SUMMARY . . . . . **vii**

TEŞEKKÜR . . . . . **viii**

İÇİNDEKİLER . . . . . **ix**

ŞEKİLLER DİZİNİ . . . . . **xii**

ÇİZELGELER DİZİNİ . . . . . **xiv**

1. GİRİŞ VE AMAÇ **1**

2. TEMEL KAVRAMLAR VE LİTERATÜR ARAŞTIRMASI **3**

2.1. Paralel Programlama Nedir? 3

2.2. Paralel Programlamanın Kullanıldığı Alanlar 5

2.3. Konsept ve Tanımlar 5

2.3.1. Von Neumann mimarisi 5

2.3.2. Flynn sınıflandırması 6

2.3.2.1. Tek komut tek veri akışı (SISD) 7

2.3.2.2. Tek komut çok veri akışı (SIMD) 7

2.3.2.3. Çok komut tek veri akışı (MISD) 8

2.3.2.4. Çok komut çok veri akışı (MIMD) 8

2.3.3. Paralel programlamada temel kavramlar 9

2.3.4. Bellek ve bilgisayar yapısı 10

2.3.4.1. Ortak bellekli 10

2.3.4.2. Dağıtık bellekli 11

2.3.4.3. Karışık bellekli 11

2.3.5. Paralel programlama tasarımı ve algoritmaları 11

2.4. Literatür Araştırması 12

3. MPI VE KÜTÜPHANELERİ **14**

3.1. MPI Kütüphaneleri 14

3.1.1. MPICH 15

3.1.2. MVAPICH2 15

3.1.3. OpenMPI 15

İÇİNDEKİLER (devam)

3.2. Gerekli Programlar . . . 15
3.3. MPICH Kurulumu . . . 15
3.3.1. Windows . . . 15
3.3.2. Linux . . . 26
3.3.3. MacOS . . . 28
3.4. MPICH'ın DevC++ ile Entegrasyonu . . . 28
3.5. Örnek Program Yapısı . . . 29
3.6. Örnek Uygulamanın Çalıştırılması . . . 30
3.6.1. Windows . . . 30
3.6.2. Linux . . . 33
3.6.3. MacOS . . . 33

4. MPICH FONKSİYONLARI . . . 35
4.1. Temel Fonksiyonlar . . . 35
4.1.1. MPI_Init . . . 35
4.1.2. MPI_Comm_Rank . . . 35
4.1.3. MPI_Comm_Size . . . 35
4.1.4. MPI_Finalize . . . 36
4.2. Noktadan Noktaya İletişim . . . 36
4.2.1. MPI_Send . . . 36
4.2.2. MPI_Recv . . . 37
4.3. Toplu İletişim . . . 47
4.3.1. MPI_Barrier . . . 47
4.3.2. MPI_Bcast . . . 48
4.3.3. MPI_Scatter . . . 50
4.3.4. MPI_Gather . . . 50
4.3.5. MPI_Allgather . . . 51
4.3.6. MPI_Reduce . . . 54
4.3.7. MPI_Allreduce . . . 55
4.3.8. MPI_Reduce_scatter . . . 55
4.3.9. MPI_Alltoall . . . 58
4.3.10. MPI_Scan . . . 59

5. CLUSTER . . . 60
5.1. Cluster Sınıfları . . . 60
5.2. Raspberry Pi Nedir? . . . 61

İÇİNDEKİLER (devam)

5.3. Cluster Oluşturma 62
5.3.1. İşletim sistemi kurulumu 62
5.3.2. MPICH kurulumu 63
5.3.3. Diğer nodlara kurulum 65
5.3.4. Nodlar arası bağlantı 66

6. SONUÇ VE ÖNERİLER 77

KAYNAKLAR DİZİNİ 78

ŞEKİLLER DİZİNİ

Şekil		Sayfa
2.1	Seri Programlama	3
2.2	Seri Programlama Örnek	3
2.3	Paralel Programlama	4
2.4	Paralel Programlama Örnek	4
2.5	Von Neumann Mimarisi	6
2.6	Tek Komut Tek Veri Akışı	7
2.7	Tek Komut Çok Veri Akışı	8
2.8	Çok Komut Tek Veri Akışı	8
2.9	Çok Komut Çok Veri Akışı	9
2.10	Düzenli Bellek Erişimi	10
2.11	Düzensiz Bellek Erişimi	10
2.12	Dağıtık Bellekli	11
3.1	Yönetici Konsolu	16
3.2	Klasör Değiştirme	17
3.3	msiexec Komutunu Çalıştırma	17
3.4	Kurulum Başlangıcı	18
3.5	Kurulum Detayları	18
3.6	Sözleşme Onayı	19
3.7	Şifre Ayarlama	19
3.8	Kurulum Yeri	20
3.9	Yeni Kurulum Yeri	20
3.10	Kurulum Başlangıcı	21
3.11	Kurulum Sonu	21
3.12	Sistem Özellikleri	22
3.13	Gelişmiş Sistem Ayarları	22
3.14	Ortam Değişkenleri	23
3.15	PATH Değişkeni Düzenleme	23
3.16	PATH Değişkeni Ekleme	24
3.17	wmpiregister Çalıştırma	24
3.18	wmpiregister Ayarları	25
3.19	smpd -status	25
3.20	MPICH İndirilmesi	26
3.21	Sıkıştırılan Dosyanın Klasöre Çıkarılması	26

ŞEKİLLER DİZİNİ (devam)

3.22 Kurulumun İlk Adımı 27
3.23 Make Komutu 27
3.24 Programın Test Edilmesi 28
3.25 Proje Oluşturma 31
3.26 Proje Ayarları 31
3.27 Linker Eklemek 31
3.28 Derleme İşlemi 32
3.29 Programın Çıktısı 32
3.30 Linux'ta Programın Çıktısı 33
3.31 MacOS'ta Programın Çıktısı 34

4.1 Uygulama 4.1 Sonucu 39
4.2 Uygulama 4.2 Sonucu 42
4.3 Uygulama 4.3 Sonucu 45
4.4 Uygulama 4 Sonucu 46
4.5 MPI_Barrier 48
4.6 MPI_Bcast 48
4.7 Uygulama 4.5 Sonucu 49
4.8 MPI_Scatter 50
4.9 MPI_Gather 51
4.10 MPI_Allgather 51
4.11 Uygulama 4.6 Sonucu 54
4.12 MPI_Reduce 54
4.13 Uygulama 4.7 Sonucu 57
4.14 Uygulama 4.8 Sonucu 58

5.1 Raspberry Pi 61
5.2 İşletim Sisteminin SD Karta Yüklenmesi 62
5.3 MPICH Kurulum Testi 65
5.4 SD Karttan Okuma 65
5.5 Kümede Uygulama Çalıştırma 67
5.6 Uygulama 5.1 Sonucu 69
5.7 Uygulama 5.2 Sonucu 71
5.8 Uygulama 5.3 Sonucu 72
5.9 Uygulama 5.4 Sonucu 74
5.10 Uygulama 5.5 Sonucu 76

ÇİZELGELER DİZİNİ

Çizelge **Sayfa**

4.1 MPI_Send Fonksiyonları 36
4.2 MPI_Reduce Fonksiyonları 55

1. GİRİŞ VE AMAÇ

Paralel programlama bir işi yapabilecek birden çok birimin aynı zamanda bir problem üzerinde çalışabilecek şekilde programlanması işlemidir. Paralel programlama ile ilgili ilk fikirler 1958 yılında John Cocke ve Daniel Slotnick tarafından ortaya atılmıştır. Bu bilgiler ileriki yıllarda üretilen süper bilgisayarların temelini oluşturmuştur. Günümüzde ise çok çekirdekli işlemcilerin geliştirilmesi ile birlikte paralel programların bilgisayarlarda kullanımı artmıştır. Paralel programlama genellikle bir bilgisayarın çözmekte zorlanacağı işlemleri birden çok bilgisayara veya işlemciye paylaştırarak çözmeyi sağlar. Bundan dolayı paralel programlama büyük veri, web arama motorları, finans modellemeleri, matematik, meteoroloji,... vb gibi geniş bir kullanım alanına sahiptir.

Paralel programlamada teorik olarak işlemleri hızlandırdığı için zamandan tasarruf sağlar. Çözümü normal bilgisayarlarla zor olan problemleri çözülebilir hale getirir. Farklı lokasyonlarda bulunan bilgisayar kümeleri aynı paralel programda kullanılabilir. Paralel programlamanın bunun gibi avantajları yanında dezavantajları da vardır. Paralel programlamada problem çözülmek için küçük parçalara ayrılır ve bu parçalar farklı bilgisayarlarda işlenir. Bu bilgisayarlardan birinde oluşabilecek sorun tüm programı etkileyebilir veya programın çözümünün hatalı çıkması ile sonuçlanabilir. Başka bir dezavantaj olarak iletişim süresi gösterilebilir. Parçalar farklı bilgisayarlarda işlendiği için bu işlemcilerin iletişiminde bir aksama tüm programa etki edecektir.

Paralel programlar dünyada ve Türkiye'de farklı amaçlarla kullanılmaktadır. Bu programlar genel olarak clusterlar üzerinde çalıştırılmaktadır. Lokasyona bağlı olmayan clusterlara SETI@home, Folding@home örnek gösterilebilir. Türkiye'de ULAKBİM'de akademik araştırmalar için kullanılabilecek bir cluster bulunmaktadır.

MPI ise paralel programlarla alakalı bir bilgisayar iletişim protokolüdür. 1994 yılında MPI'ın birinci versiyonu Message Passing Interface Forum tarafından piyasaya sürülmüştür.

MPI bir iletişim protokolü olduğu için farklı yazılım dilleri ile kullanılabilmesi kütüphaneler yardımı ile sağlanmaktadır. Bu tez çalışmasında MPICH kütüphanesi kullanılmıştır.

Bu tez çalışması genel anlamda paralel programlamanın ne anlama geldiği, MPI ile paralel programlar geliştirmek için gerekli kütüphanenin kullanımı, bu kütüphanenin fonksiyonları hakkında bilgiler içermektedir.

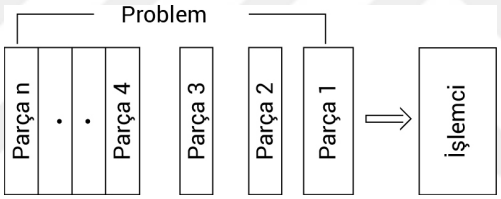
Aynı zamanda bu bilgileri kullanarak paralel programlar yazılması ve paralel programlama ile seri programlamanın performans farkının ölçülmesi amaçlanmaktadır.

2. TEMEL KAVRAMLAR VE LİTERATÜR ARAŞTIRMASI

Bu bölümde paralel programlama ile ilgili temel bilgiler ve literatür araştırması verilecektir.

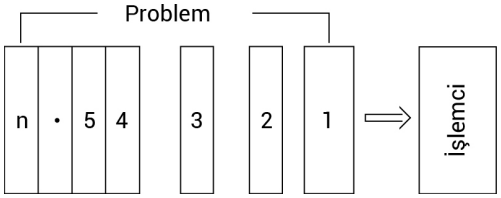
2.1 Paralel Programlama Nedir?

Paralel programlamanın kolayca anlaşılabilmesi için öncelikle yazılımda seri programlama olarak adlandırılan standart programlama metodunun anlaşılması gerekmektedir. Seri programlama mantığında problem küçük parçalara ayrılır. Parçalar tek bir işlemci üzerinde sırayla işlenir. (Şekil 2.1)



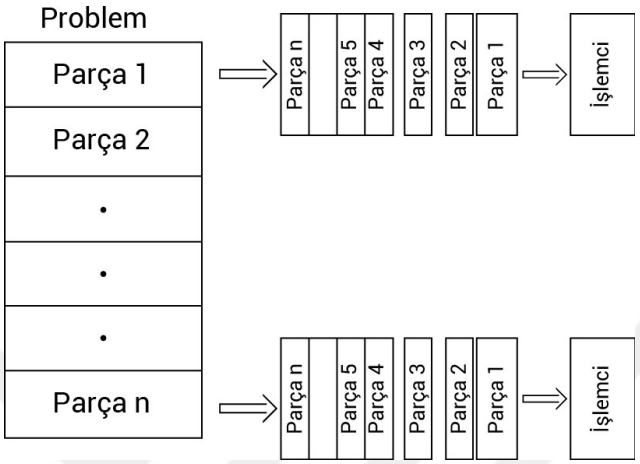
Şekil 2.1 Seri Programlama

Örnek olarak 1'den n'ye kadar olan sayıların toplamını alacak bir program geliştirilirse, sayılar 1'den n'ye kadar sıralanır. Toplam sıfır alınır ve 1'den başlanarak n'ye kadar tüm sayılar toplama eklenir. (Şekil 2.2)



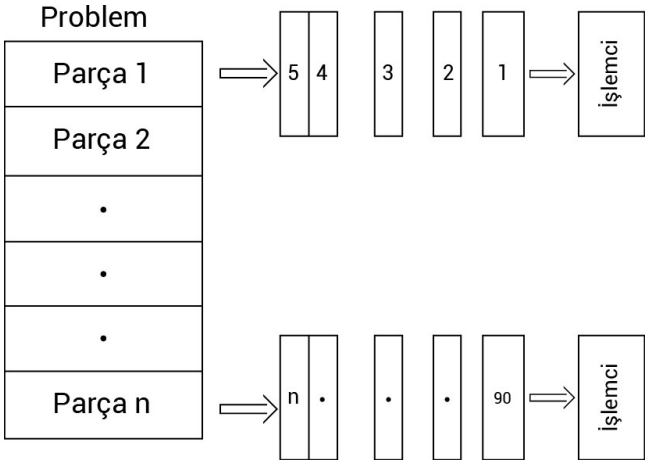
Şekil 2.2 Seri Programlama Örnek

Paralel programlamada ise öncelikle problem farklı işlemcilere gönderilmek için parçalara ayrılır. Daha sonra ayrılan parçalar seri programlama mantığında olduğu gibi farklı işlemcilerde bir daha küçük parçalara ayrılır ve bu parçalar işlemci üzerinde sırayla işlenir. (Şekil 2.3)



Şekil 2.3 Paralel Programlama

Seri programlama mantığındaki örnek, paralel programlamaya uygulandığında, öncelikle sayılar işlemci sayısına göre parçalara bölünür. Ardından seri programlama mantığındaki gibi bu bölünen parçalar da tek bir işlemcide işlenecek şekilde daha küçük parçalara bölünür ve sırayla işlenir. (Şekil 2.4)



Şekil 2.4 Paralel Programlama Örnek

Paralel programlamanın genel olarak tercih edilme sebepleri şunlardır:

- Teorik olarak problem parçaları aynı anda farklı işlemciler üzerinde çalıştığı için zamandan tasarruf sağlar.
- Standart programlama ile tek işlemci üzerinde çözülmesi uzun zaman alan problemlerin çözümünü kolaylaştırır.
- Modern bilgisayarlarda bulunan çok çekirdekli işlemcilerin tam verimli olarak çalışmasını sağlar.
- Farklı lokasyonlarda bulunan bilgisayar kümeleri aynı paralel programda kullanılabilir.

2.2 Paralel Programlamanın Kullanıldığı Alanlar

Paralel programlama büyük veri, veri tabanları, veri madenciliği, web arama motorları, SaaS uygulamaları, hastalık teşhisleri, eczacılık, finans modellemeleri, ekonomik modellemeler, 3D modellemeler, sanal gerçeklik, uygulamalı fizik, nükleer fizik, parçacık fiziği, bioteknoloji, genetik, jeoloji, makine mühendisliği, mikroelektronik, bilgisayar bilimleri, matematik, bilimsel araştırmalar, meteoroloji,... gibi geniş bir kullanım alanına sahiptir.

Aynı zamanda lokasyona bağlı olmadan oluşturulan bilgisayar kümeleri vardır. Bu kümelere boşta olan bilgisayarlar bağlanarak araştırmalara destek olunabilir. Bu kümelere SETI@home, Folding@home örnek olarak gösterilebilir. Türkiye'de de ULAKBİM'de akademik araştırmalar için kullanılabilecek bir bilgisayar kümesi bulunmaktadır.

2.3 Konsept ve Tanımlar

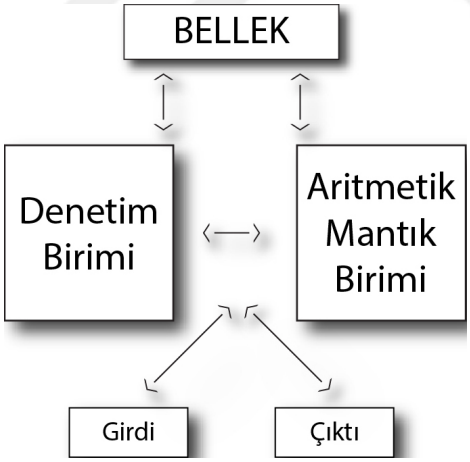
2.3.1 Von Neumann mimarisi

Von Neumann mimarisi matematikçi ve bilgisayar bilimci John von Neumann'ın 30 Haziran 1945 tarihli makalesine dayanır (Von Neumann, 1945). Paralel bilgisayarlar bu mimariyi kullanırlar.

Mimari dört bileşenden oluşur. Bu bileşenler,

1. Bellek
2. Denetim Birimi
3. Aritmetik Mantık Birimi
4. Girdi / Çıktı

Bellek program komutlarını ve verilerini tutar. Denetim birimi program komutları ve veriler arasındaki iletişimi koordine eder. Aritmetik mantık birimi basit aritmetik hesapları yapar. (Şekil 2.5) Von Neumann mimarisi iki sorunu beraberinde getirmiştir. Bu sorunlardan ilki çökme sorunudur. Hatalı yazılan programlar yanıt vermeyi durdurabilirler hatta bilgisayarı kitleyebilirler. İkinci sorun darboğaz sorunudur. 1977 yılında John Backus tarafından yayınlanmıştır. Darboğaz bileşenler arasındaki uyumsuzluk sorunudur. Örneğin işlemcilerin hız artışları belleklerin hız artışını geçtiği için işlemci işlemi yapsa dahi bellek bunu yazamamaktadır (Backus, 1978).



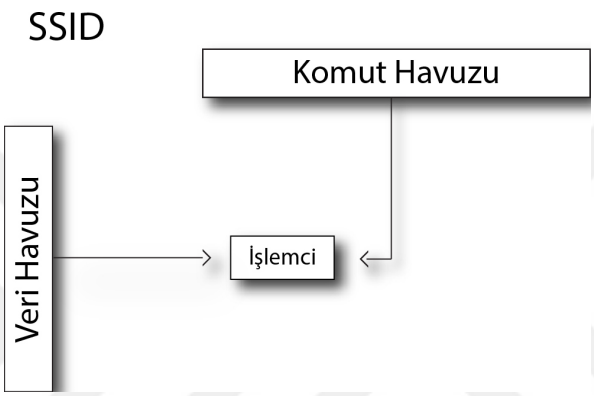
Şekil 2.5 Von Neumann Mimarisi

2.3.2 Flynn sınıflandırması

1966 yılında Michael J. Flynn tarafından oluşturulan bilgisayar mimarisi sınıflandırmasıdır (Flynn, 1972). Flynn sınıflandırmasına göre dört ana sınıf vardır. Bu sınıflar tek komut tek veri akışı, tek komut çok veri akışı, çok komut tek veri akışı ve çok komut çok veri akışıdır.

2.3.2.1 Tek komut tek veri akışı (SISD)

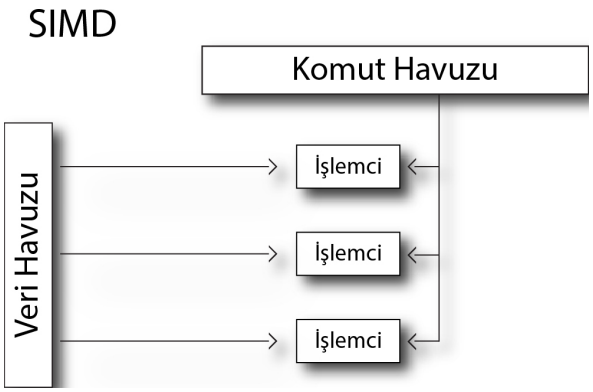
- Seri bilgisayarlardır. (Şekil 2.6)
- Birim zamanda tek komut işlenir.
- Birim zamanda tek veri akışı kullanılır.
- Von Neumann mimarisine karşılık gelmektedir.



Şekil 2.6 Tek Komut Tek Veri Akışı

2.3.2.2 Tek komut çok veri akışı (SIMD)

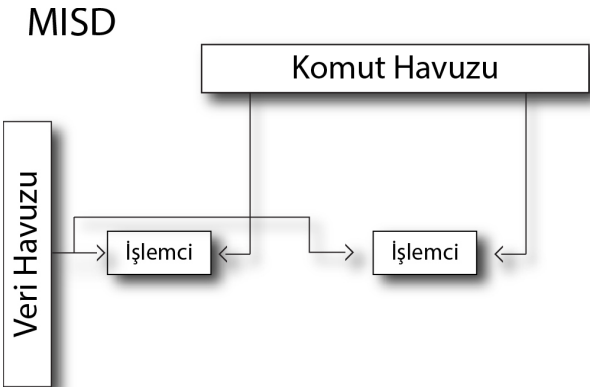
- Paralel bilgisayarlardır. (Şekil 2.7)
- Birim zamanda tek komut işlenir.
- Birim zamanda birden çok veri akışı kullanılır.
- İşlemler senkronize yapılır.



Şekil 2.7 Tek Komut Çok Veri Akışı

2.3.2.3 Çok komut tek veri akışı (MISD)

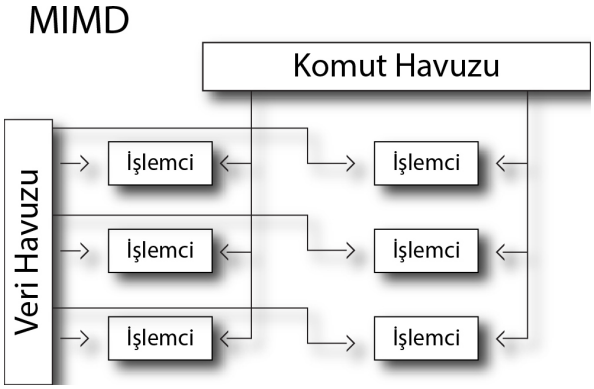
- Paralel bilgisayarlardır. (Şekil 2.8)
- Birim zamanda birden çok komut işlenir.
- Birim zamanda tek veri akışı kullanılır.



Şekil 2.8 Çok Komut Tek Veri Akışı

2.3.2.4 Çok komut çok veri akışı (MIMD)

- Paralel bilgisayarlardır. (Şekil 2.9)
- Birim zamanda birden çok komut işlenir.
- Birim zamanda birden çok veri akışı kullanılır.



Şekil 2.9 Çok Komut Çok Veri Akışı

2.3.3 Paralel programlamada temel kavramlar

Bu bölümde paralel programlamada kullanılan temel kavramlar verilecektir.

Node : Bir bilgisayar kümesindeki tek bir bilgisayarı ifade eder.

CPU : Çalıştırılmakta olan yazılımın komutlarını işleyen birimdir.

Task : İşlemci tarafından işlenen program veya program komut grubudur.

Pipelining : Komutların farklı işlemcilerde işlenmek üzere parçalara bölünmesidir.

Shared memory : Tüm işlemcilerin aynı fiziksel belleğe erişiminin olmasıdır.

Distributed memory : Tüm işlemcilerin kendi belleklerine erişimi olmasıdır.

Communications : Paralel programlamada komutların girdi ve çıktı alımları aşamasında kullanılmasıdır.

Parallel overhead : Komutların koordine bir şekilde çalışması için gerekli olan zamandır.

Synchronization : Komutların gerçek zamanlı olarak koordine edilmesidir.

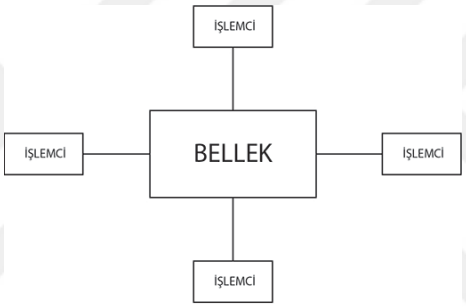
Scalability : Üretilen paralel sistemin daha fazla kaynak eklenerek hızının arttırılmasını belirtir. Donanım, algoritma değişimleri gibi farklı metodlarla yapılabilir.

2.3.4 Bellek ve bilgisayar yapısı

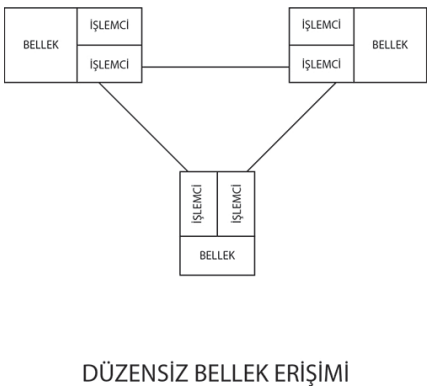
Bellek, bilgi depolama ünitesidir. Belleklerde saklanan tüm bilgiler mantıksal olarak 0 ve 1'lerden oluşurlar. Belleğin yapısına göre bu bilgiler kısa veya uzun süre depolanabilir. Paralel bilgisayarlarda üç çeşit bellek yapısı kullanılır. Bunlar ortak bellekli, dağıtık bellekli ve karışık bellekli paralel bilgisayarlardır.

2.3.4.1 Ortak bellekli

Genel olarak tüm işlemcilerin tüm hafızaya erişimi olduğu paralel bilgisayar türüdür. Düzenli ve düzensiz bellek erişimi olarak ikiye ayrılır. Düzenli bellek erişimli bilgisayarlarda tüm işlemcilerin belleğe erişim zamanı eşittir. (Şekil 2.10) Düzensiz bellek erişiminde ise bu zaman belleğin işlemci üzerindeki yerine bağlıdır. (Şekil 2.11)



Şekil 2.10 Düzenli Bellek Erişimi



Şekil 2.11 Düzensiz Bellek Erişimi

2.3.4.2 Dağıtık bellekli

Dağıtık bellekli sistemlerin genel özelliği bellekleri ve işlemcileri bağlamak için bir iletişim ağına ihtiyaç vardır. İşlemcilerin kendilerine ait bellekleri vardır. Tüm işlemcilerin kendilerine ait bellekleri olduğu için diğerlerinden bağımsız bir şekilde işlem görebilirler. Veri akışları sırasında iletişim ağı kullanılır. (Şekil 2.12)



Şekil 2.12 Dağıtık Bellekli

2.3.4.3 Karışık bellekli

Ortak ve dağıtık bellek aynı sistemde bulunur. Ortak bellekli bölüm ortak bellekli makine veya grafik işlemci olabilir. Dağıtık bellekli bölüm ortak bellekli makineler veya grafik işlemciler arasındaki iletişim ağından oluşur. Günümüzde paralel sistemlerde bu yapı kullanılır.

2.3.5 Paralel programlama tasarımı ve algoritmaları

Paralel programlar farklı şekillerde programlanabilir. Bunlardan bazıları şunlardır:

- Ortak Bellekler
- Thread
- Dağıtık Bellekler / İleti Temelli İletişim
- Karışık Bellekler
- Program kodlarının paralelleştirilmesi

Günümüzde kullanılan seri programlama mantığında her bir adımda bir komut işlenmektedir. Bu şekilde kullanılan algoritmalar paralel programlama açısından problemlidir. Bu problemi çözebilmek için paralel programlama algoritmaları

tasarlanmıştır. Bu algoritmalara böl ve yönet algoritması örnek gösterilebilir. “Böl ve Yönet” eski uygarlıklar tarafından ülkelerini idare etmek veya yeni yerler fethetmek için kullanılan tekniktir. Bu teknik büyük toplulukların küçük parçalara ayrılarak idare edilmesi veya fethedilmesini kolaylaştırır. Paralel algoritma olarak ise elde bulunan problemi küçük alt bölümlere bölerek daha kolay çözülmesi hedeflenir. Bu küçük bölümler programın belirteceği işlemleri yaptıktan sonra çözümleri geri döndürürler. Bu çözümler birleştirilerek asıl problemin çözümü bulunur. Problem küçük bölümlere bölündüğü için idaresi kolaylaşır ve bu bölümler paralel olarak çözülebilir (Blelloch, Maggs, 1996).

2.4 Literatür Araştırması

Paralel programlama ile alakalı ilk fikirler John Cocke ve Daniel Slotnick tarafından 1958 yılında bir IBM araştırmasında ortaya atılmıştır. Daniel Slotnick bu fikirle SOLOMON adında bir bilgisayar tasarlamış, fakat hayata geçirilememiştir (Schneck, 1987). Bu fikirler ve tasarımlar 1960 ve 1970’li yıllarda üretilen süper bilgisayarların temelini oluşturmuştur.

1960 yılında ise Novosibirsk Matematik Enstitüsün’den E. V. Yevreinov paralel mimarilerin bağlantılı bir şekilde programlanmasını geliştirmiştir.

1964 yılında Livermore National Laboratuarında kullanılmak üzere paralel makineler Daniel Slotnick tarafından geliştirilmiştir (Slotnick, 1982).

1967 yılında AFIPS Konferansında paralel işlemlerle ilgili bir makale Gene Amdahl ve Daniel Slotnick tarafından yayınlanmıştır. Bu makale ile ilgili tartışmalar ”Amdahl Kanunu” olarak adlandırılmıştır (Amdahl, 1967).

1983 yılında Goodyear Aerospace firması Goddard Uzay Uçuş Merkezi için MPP’yi (Massively Paralel Processor) geliştirilmiştir (Batcher, 1980).

1980’li yılların sonlarına doğru ise MPP’lerin yerlerini clusterlar almıştır.

1985 yılında Linda paralel programlama protokolünün temelleri David Gelernter tarafından atılmıştır (Gelernter, 1985).

1986 yılında PVM (Parallel Virtual Machine) dağıtık bilgisayarlar için geliştirilmiştir (Akçay, Erdem, 2010).

1990 yılında Massachusetts Institute of Technology üniversitesinde paralel programlamaya yönelik Cilk dili geliştirilmiştir (Anonim, 1990).

1993 yılında IBM ilk SP1 Powerparallel platformunu piyasaya sürmüştür. Platformun SP2 modeli 1995, SP3 1999 yıllarında piyasaya sürülmüştür (Franke, 1995).

Günümüzde ise çok çekirdekli işlemcilerin geliştirilmesi ile birlikte paralel programlamanın kullanıldığı alanlar artmıştır. Seri programlama işlemci hızına bağlı iken paralel programlama hem işlemci hem de işlemci çekirdek sayısına bağlıdır. İşlemci hızını arttırmak işlemciye çekirdek eklemekten daha maliyetli olduğundan paralel programlamanın daha avantajlı hale geldiği görülmüştür. Bunlar bu artışın sebebi olarak gösterilebilir

3. MPI VE KÜTÜPHANELERİ

MPI (Message Passing Interface), çok karıştırılanın aksine bir dil veya derleyici değil, bilgisayar iletişim protokolüdür. MPI'ın temelleri 1991 yılında Avusturya'da bir grup araştırmacı tarafından atılmıştır. 1992 yılında ise Jack Dongarra, Tony Hey ve David W. Walker tarafından MPI1'in ilk taslağı yayınlanmıştır. 1994 yılında ise MPI'ın birinci versiyonu piyasaya sürülmüştür (MPI Forum, 1994).

MPI'ın birinci versiyonunda paylaşımlı bellek modeli bulunmamaktadır. MPI'ın ikinci versiyonunda bu özellik kısıtlı olarak eklenmiştir. MPI'ın ikinci versiyonu 2009 yılında piyasaya sürülmüştür (Message Passing Interface Forum, 2009).

MPI'ın ikinci versiyonu ise 2012 yılında piyasaya sürülmüştür (MPI Forum, 2012).

MPI dağıtık bellekli sistemlerde nodelar(nod) arasındaki iletişimde kullanılır. Bu iletişim nodlar arasında mesajların alınması ve verilmesi ile sağlanır. Her mesajın kendine özel benzersiz bir etiketi bulunur. Alıcı nod bu etiketi kontrol ederek (veya etmeden) mesajı alır. Bu işlem noktadan noktaya iletişim olarak bilinir. Bu iletişim türünün dışında bir nodun diğer nodlara yayın yaptığı toplu iletişim de vardır.

MPI'ın standart, taşınabilir, fonksiyonel, kullanımının esnek olması tercih edilmesindeki başlıca sebepler olarak gösterilebilir.

MPI programları C, C++, Python vb. diller ile yazılan kütüphanelerle oluşturulur. Genel olarak bir MPI programı, kütüphaneleri yükleyen header ve MPI fonksiyonlarından oluşur.

3.1 MPI Kütüphaneleri

MPI bir iletişim protokolü olduğu için farklı yazılım dilleri ile kullanılabilmesi kütüphaneler yardımı ile sağlanmaktadır. MPI'ın birçok kütüphanesi bulunmasına rağmen MPICH, MVAPICH2, OpenMPI bu kütüphanelerden popüler olanlarıdır.

3.1.1 MPICH

MPICH 1992 yılında geliştirilmiş olup mpich.org üzerinden Windows, macOS ve Linux işletim sistemleri için gerekli versiyonları indirilebilir. MPI-1, MPI-2, MPI-2.1, MPI-2.2 ve MPI-3 standartlarını desteklemektedir.

3.1.2 MVAPICH2

Paralel programlar yazabilmek için gerekli olan farklı programları içerir. Ohio Üniversitesi başta olmak üzere birçok dernek ve firmanın destekleriyle çalışmasına devam etmektedir. MPI-3.1 standartını desteklemektedir.

3.1.3 OpenMPI

Akademik, özel kuruluş ve bireysel destekçilerle geliştirilen bir kütüphanedir. MPI-3.1 standartını desteklemektedir.

Çalışılan tüm işletim sistemlerini desteklediği, kurulumunun kolay olması ve dökümanının gelişmiş olması sebebiyle bu çalışmada kullanılmak üzere MPICH kütüphanesi seçilmiştir. Çalışma boyunca bu kütüphanenin desteklediği C dili ile paralel programlar yazılmıştır.

3.2 Gerekli Programlar

MPI ile çalışabilmek için öncelikle MPI iletişim protokolü ile çalışabilen bir kütüphaneye ihtiyaç vardır. Daha önceki bölümlerde anlatıldığı üzere çalışmaya uygun olan MPICH kütüphanesi bu amaçla seçilmiştir. İleriki bölümlerde bu kütüphanenin Windows, Linux ve macOS işletişim sistemlerine kurulumları gösterilecektir.

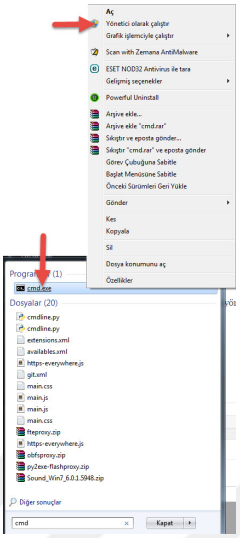
Yazılmış olan programlar; daha kolay takip edilmesi ve hataların anlık olarak bildirilmesi sebebiyle bir editörde yazılabilir. Bu adım zorunlu olmamakla beraber çalışmayı hızlandırdığı için tavsiye edilmektedir. Bu çalışmada Dev C++ editörü kullanılmıştır ve bu editörün MPICH ile entegrasyonu anlatılmıştır.

3.3 MPICH Kurulumu

3.3.1 Windows

Öncelikle MPICH programı http://www.mpich.org/static/downloads/ web adresinden bilgisayara indirilir. Bu çalışmada 1.4.1p1 klasöründe bulunan 1.4.1 versiyonu

kullanılmıştır. Klasörün içinden bilgisayara uygun olan versiyonu indirilir. Kuruluma başlamadan önce windows konsolu yönetici olarak çalıştırılır. (Şekil 3.1)



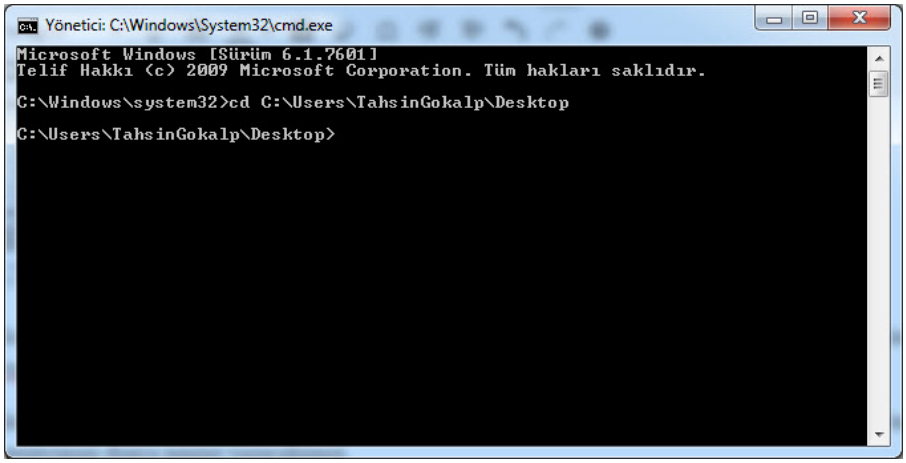
Şekil 3.1 Yönetici Konsolu

Windows işletim sisteminde MPICH kurulumu yapıldıktan sonra çalışmama, çalıştığı halde hata verme gibi problemlerle karşılaşılabilir. Bundan dolayı eğer daha önce bilgisayara MPICH kurulumu yapıldıysa silinmesi tavsiye edilmektedir.

Kurulum dosyasının bulunduğu klasöre geçiş yapılır. İndirilen dosya masaüstünde bulunduğu için

```
1 cd C:\Users\TahsinGokalp\Desktop
```

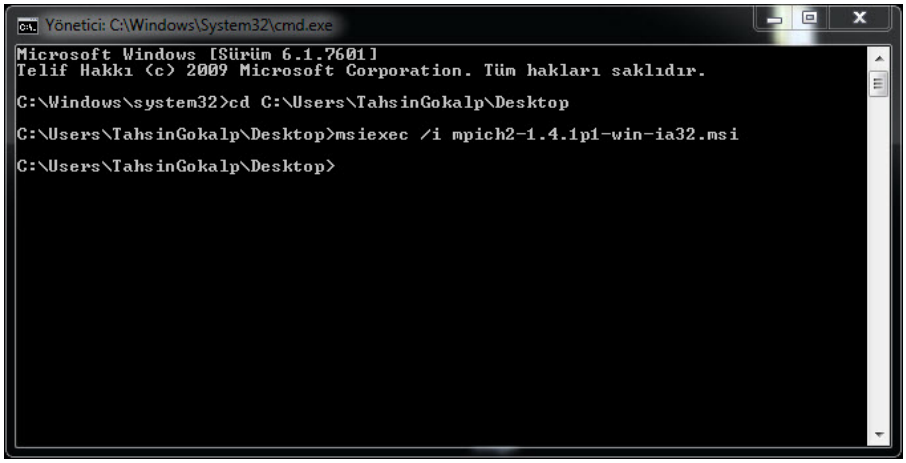
komutu kullanılmıştır. (Şekil 3.2)



Şekil 3.2 Klasör Değiştirme

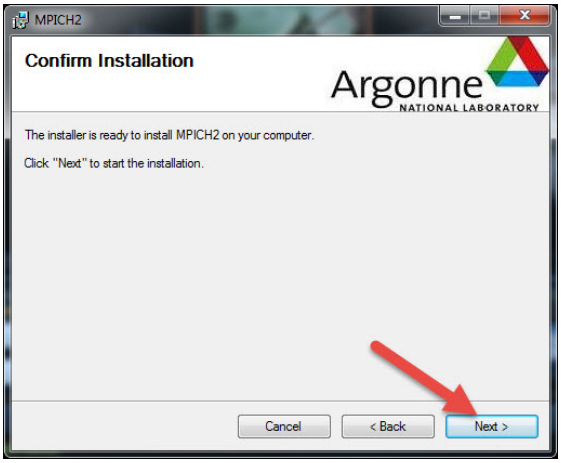
```
msiexec /i mpich2-1.4.1p1-win-ia32.msi
```

komutunu kullanarak kurulum başlatılır. **'mpich2-1.4.1p1-win-ia32.msi'** yerine bilgisayara indirilen versiyonun dosya ismi yazılmalıdır. (Şekil 3.3)



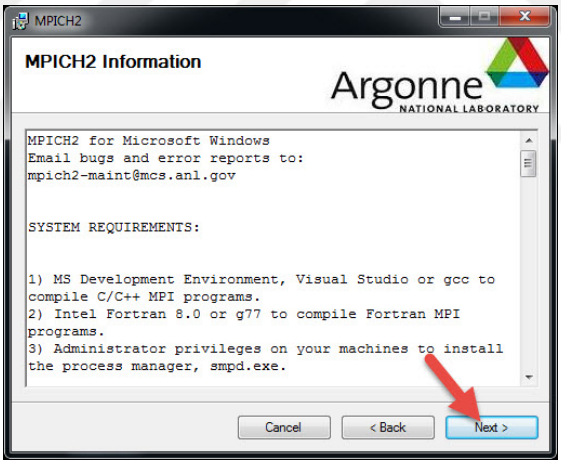
Şekil 3.3 msiexec Komutunu Çalıştırma

İlk adımda yapılacak birşey olmadığı için 'Next'e basarak kurulum başlatılır. (Şekil 3.4)



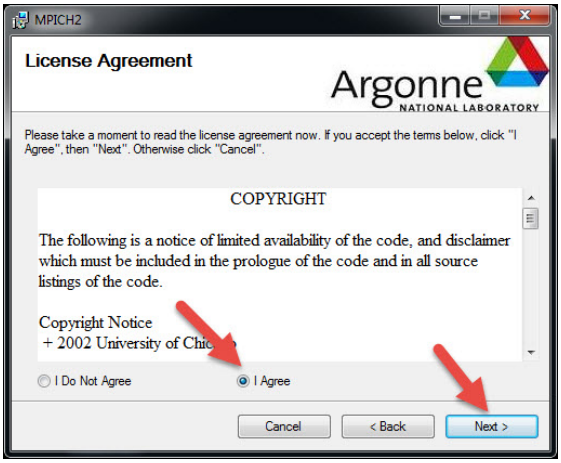
Şekil 3.4 Kurulum Başlangıcı

Bu adımda MPICH kullanımı ile ilgili bilgiler gösterilmektedir. Bu adım da ’Next’e basarak geçilir. (Şekil 3.5)



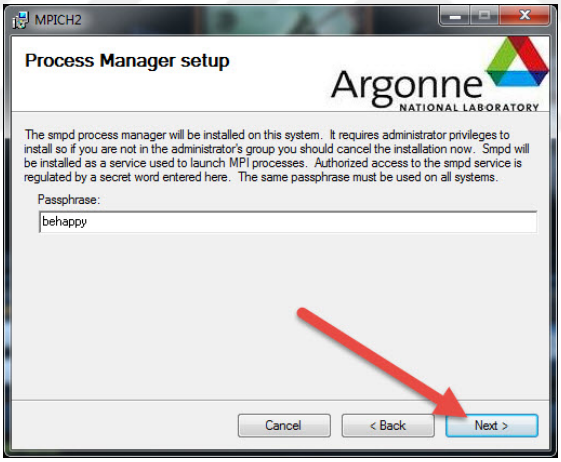
Şekil 3.5 Kurulum Detayları

Bu adımda MPICH programının sözleşmesi okunup ’Next’e basılarak devam edilir. (Şekil 3.6)



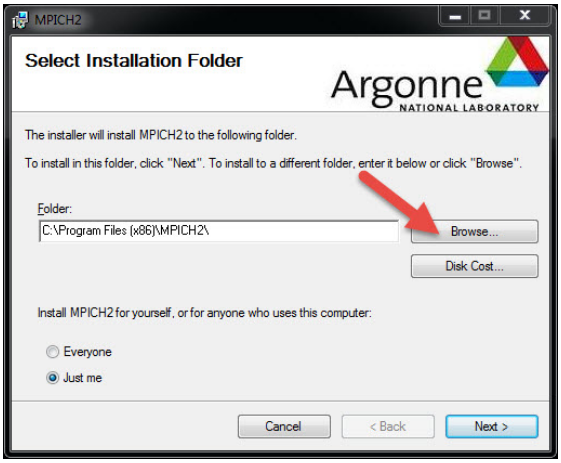
Şekil 3.6 Sözleşme Onayı

Bu adımda bilgisayara giriş yapılan şifrenin şifre kutusuna girilmesi gerekmektedir. Şifre yoksa bu şekilde bırakılıp devam edilir. (Şekil 3.7)



Şekil 3.7 Şifre Ayarlama

Bu adımda MPICH programının bilgisayara kurulduğu klasörün değiştirilmesi gerekmektedir. Browse butonuna basılır. (Şekil 3.8)

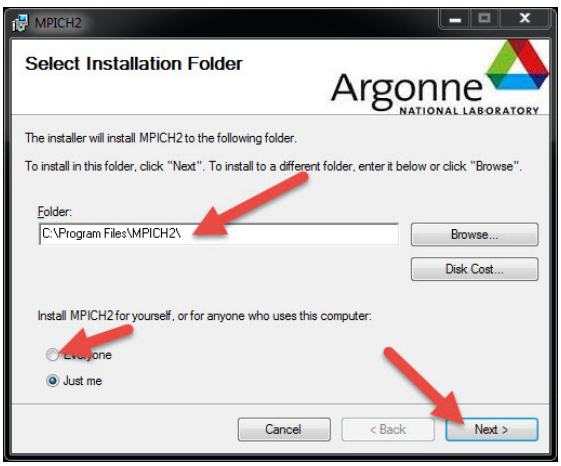


Şekil 3.8 Kurulum Yeri

Ardından kurulum klasörü aşağıdaki şekilde belirlenir.

```
C:\Program Files\MPICH2\
```

belirlenir ve ‘Everyone’ seçeneği işaretlenerek ‘Next’ butonuna basılır. (Şekil 3.9)



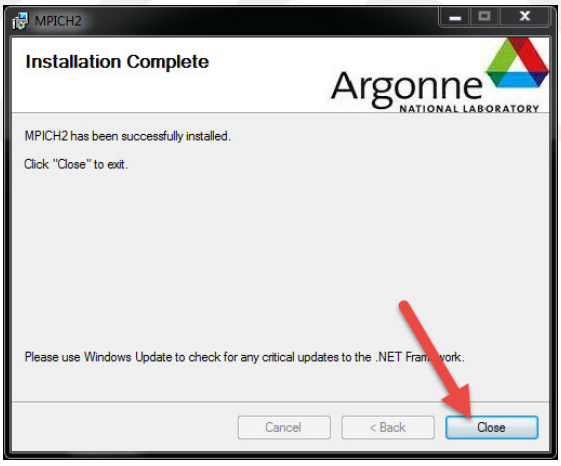
Şekil 3.9 Yeni Kurulum Yeri

Bu adımda ‘Next’ butonuna basarak kurulum başlatılır. (Şekil 3.10)



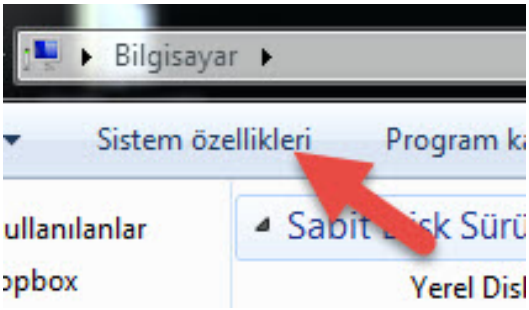
Şekil 3.10 Kurulum Başlangıcı

Program dosyaları bilgisayara kopyalandıktan sonra 'Close' butonuna basarak kurulum tamamlanır. (Şekil 3.11)



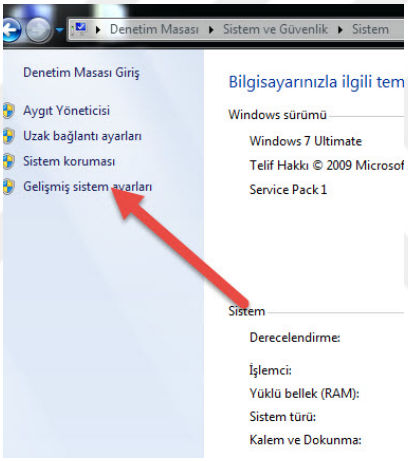
Şekil 3.11 Kurulum Sonu

Kurulum tamamlandıktan sonra MPICH programının çalışması için MPICH dosyalarının Windows Ortam Değişkenlerine eklenmesi gerekir. Bunun için Bilgisayarımdan Sistem Özelliklerine girilir. (Şekil 3.12)



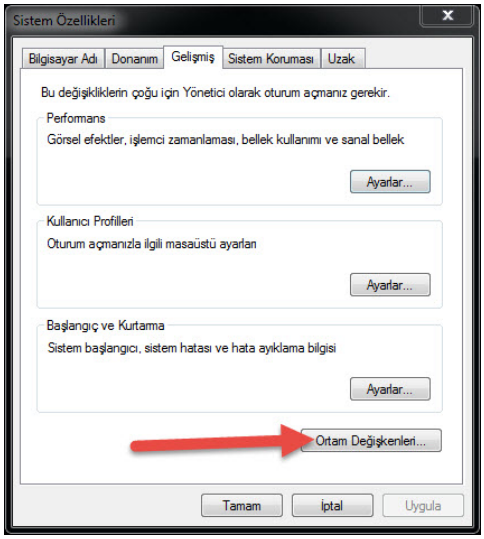
Şekil 3.12 Sistem Özellikleri

Gelişmiş sistem ayarlarına girilir. (Şekil 3.13)



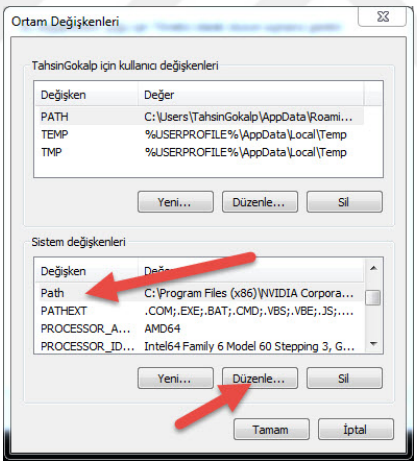
Şekil 3.13 Gelişmiş Sistem Ayarları

Ortam değişkenlerine girilir. (Şekil 3.14)



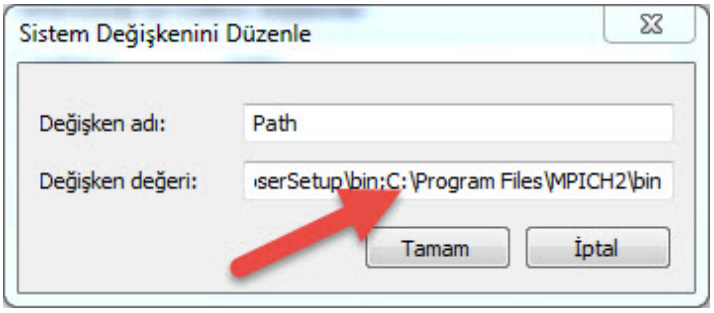
Şekil 3.14 Ortam Değişkenleri

Path değişkeni düzenlenir. (Şekil 3.15)



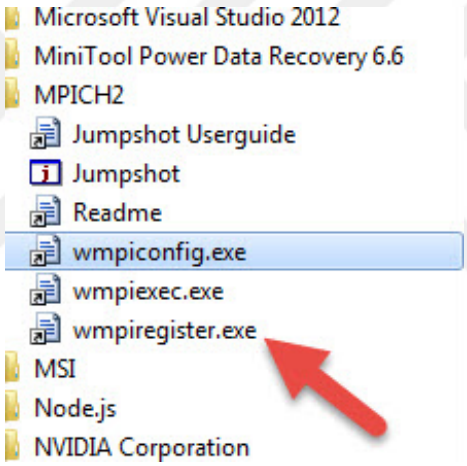
Şekil 3.15 PATH Değişkeni Düzenleme

MPICH programının kurulduğu klasörün sonuna bin eklenerek listenin en sonuna dahil edilir. (Şekil 3.16)



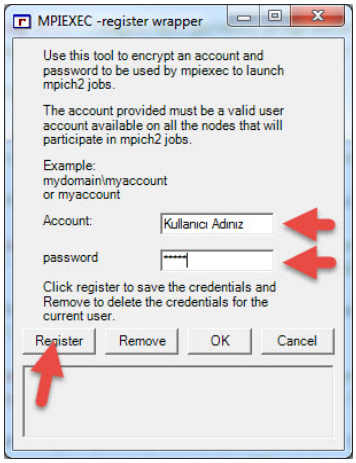
Şekil 3.16 PATH Değişkeni Ekleme

Path değişkenine gerekli ekleme yapıldıktan sonra MPICH programına yönetici hesabının tanıtılması gerekir. Bunun için Başlat'tan wmpiregister.exe açılır. (Şekil 3.17)



Şekil 3.17 wmpiregister Çalıştırma

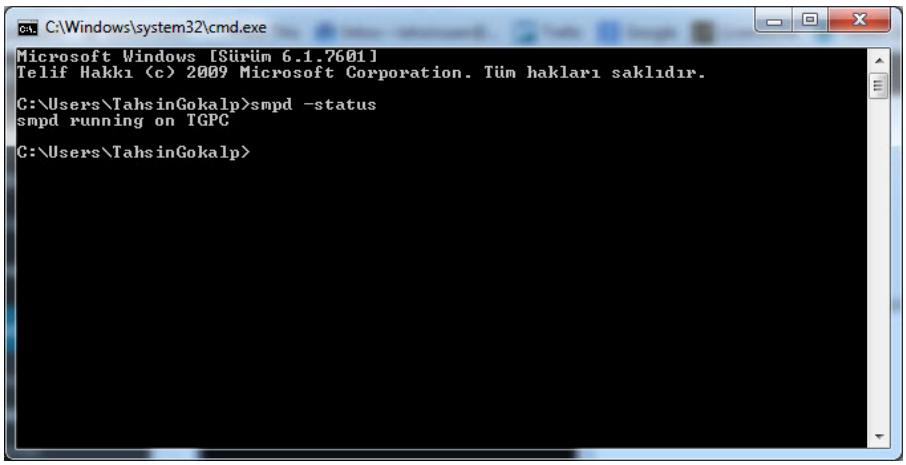
Açılan programda 'Account' alanına bilgisayara giriş yapılan kullanıcı adı yazılır. 'password' alanına bilgisayarın şifresi yazılır ve 'Register' butonu ile yönetici hesabı MPICH programına kaydedilir. (Şekil 3.18)



Şekil 3.18 wmpiregister Ayarları

MPICH programının problemsiz bir şekilde çalışıp çalışmadığının kontrol edilmesi için yönetici olarak bir konsol penceresi açılır aşağıdaki kod ile kontrol edilir. (Şekil 3.19)

```
1 smpd -status
```



Şekil 3.19 smpd -status

Yukarıdaki gibi bir çıktı başarılı bir kurulumun sonucudur.

3.3.2 Linux

Kurulum Ubuntu 16.04 üzerinde yapılmıştır. MPICH programını http://www.mpich.org/static/downloads/ web adresinden bilgisayara indirilir. Bu çalışmada 1.4.1p1 klasöründe bulunan 1.4.1 versiyonu kullanılmıştır.

```
1 wget http://www.mpich.org/static/downloads/1.4.1p1/mpich2-1.4.1p1.tar.gz
```

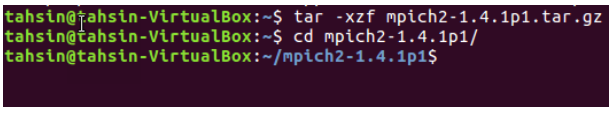
Yukarıdaki komut ile çalışmada kullanılan versiyonu bilgisayara indirilir. (Şekil 3.20)



Şekil 3.20 MPICH İndirilmesi

```
1 tar -xzf mpich2-1.4.tar.gz
2 cd mpich2-1.4
```

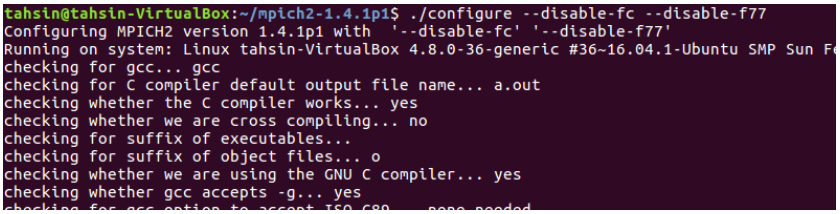
Yukarıdaki komutlar ile sıkıştırılmış dosya klasöre çıkarılır ve o klasöre geçiş yapılır. (Şekil 3.21)



Şekil 3.21 Sıkıştırılan Dosyanın Klasöre Çıkarılması

```
1 ./configure --disable-fc --disable-f77
```

Yukarıdaki komut ile kurulumun ilk adımına başlanır. (Şekil 3.22)

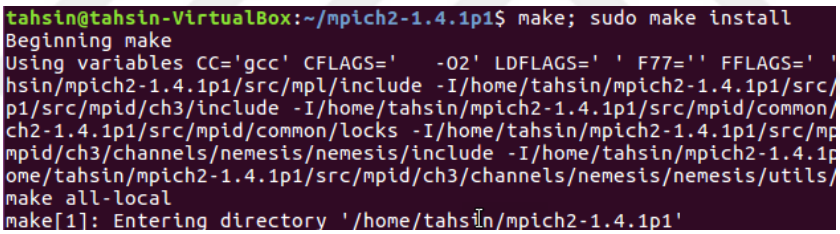


Şekil 3.22 Kurulumun İlk Adımı

"Configuration completed." mesajı görüldükten sonra yükleme dosyası oluşturulup programın yüklenmesi için

```
1 make; sudo make install
```

komutu çalıştırılır. (Şekil 3.23)



Şekil 3.23 Make Komutu

```
1 mpiexec --version
```

komutu ile MPICH programının bilgisayara problemsiz bir şekilde yüklenip yüklenmediği kontrol edilir. (Şekil 3.24)



Şekil 3.24 Programın Test Edilmesi

3.3.3 MacOS

```
1 brew install mpich2
```

komutu ile MacOS üzerinde homebrew yardımı ile mpich kurulumunu tamamlanır.

3.4 MPICH'ın DevC++ ile Entegrasyonu

Dev C++ ile MPICH entegrasyonu yapılarak hızlı bir şekilde proje oluşturulabilir. Bunun için editöre bir template dosyası eklenmelidir. Bu template dosyası Dev C++'ın kurulum klasörünün içinde bulunan Templates klasöründe oluşturulmalıdır.

Bu çalışmada Dev C++'ın kurulu olduğu klasör C:\Program Files (x86)\Dev-Cpp\Templates olduğu için bu klasörün içine MPICH.template ve mpich.txt adında iki dosya oluşturulur.

MPICH.template dosyasının içeriği aşağıdaki gibi olmalıdır.

```
1 [Template]
2 ver =1
3 Name =MPICH
4 Icon =Communication.ico
5 Description =MPICH proje
6 Category =Console
7
8 [Unit0]
9 CName =main.c
10 C =mpich.txt
11
```

```
[Project]
UnitCount =1
Type =1
Icon =Communication.ico
```

mpich.txt dosyasının içeriği aşağıdaki gibi olmalıdır.

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

#define ROOT 0

int main(int argc, char **argv) {

int myrank, nprocs;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

// TODO: Parallel code here
printf("Hello from processor %d of %d\n", myrank, nprocs);

MPI_Finalize();

return 0;
}
```

Bu iki dosyanın eklenmesi ile birlikte entegrasyon tamamlanmış olur. İlk proje için File -> New -> Project menüsüne erişilir. Console menüsü altında MPICH seçmi yapılarak ilk projemiz oluşturulur.

3.5 Örnek Program Yapısı

MPI'ı daha iyi açıklamak için aşağıdaki örnek kod incelenebilir.

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

#define ROOT 0

int main(int argc, char **argv) {
int myrank, nprocs;
```

```
 8
 9 MPI_Init(&argc, &argv);
10 MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
11 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
12
13 //Paralel program kodları
14
15 MPI_Finalize();
16
17 return 0;
18 }
```

Öncelikli olarak **#include <mpi.h>** başlık dosyası programa eklenir. Ardından MPI'ı başlatacak olan **MPI_Init(&argc, &argv);** fonksiyonu yazılır. Bu iki satır her MPI programında bulunur.

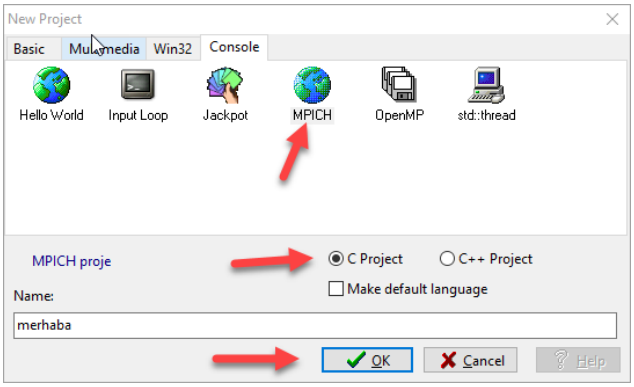
MPI_Comm_size(MPI_COMM_WORLD, &nprocs); o anda programı çalıştıran toplam işlemci sayısını verir. **MPI_Comm_rank(MPI_COMM_WORLD, &myrank);** programın üzerinde çalıştığı işlemcinin sırasını belirtir. Bu sıra sıfırdan başlayarak işlemci sayısı - 1'e kadar devam eder.

MPI_Finalize(); ise MPI programının tamamlandığını belirtir. Bundan sonra yazılacak kodlar paralel olarak çalıştırılmaz. Genel olarak MPI programları bu şekilde oluşturulur.

3.6 Örnek Uygulamanın Çalıştırılması

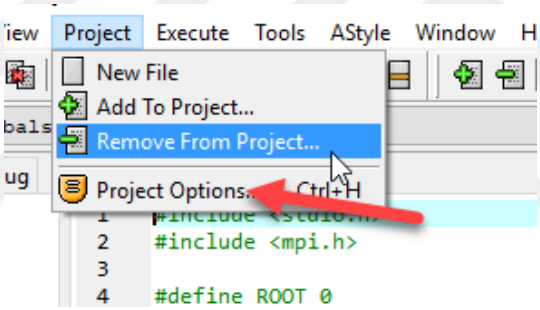
3.6.1 Windows

Daha önceki bölümlerde Dev C++ ile MPICH entegrasyonu yapılmıştı. Bu entegrasyon kullanılarak örnek bir proje aşağıdaki sıra takip edilerek oluşturulur. (Şekil 3.25)



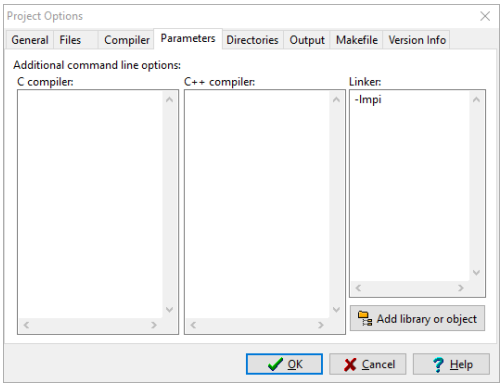
Şekil 3.25 Proje Oluşturma

Projenin sorunsuz bir şekilde derlenmesi için bazı ayarlar yapılmalıdır. Project menüsü altında ki Project Options bölümüne giriş yapılır. (Şekil 3.26)



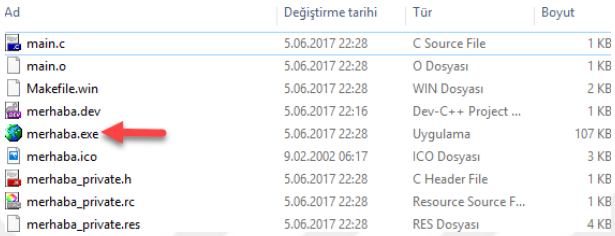
Şekil 3.26 Proje Ayarları

Burada Parameters menüsünde ki linkler bölümüne -lmpi komutu eklenir. (Şekil 3.27)



Şekil 3.27 Linker Eklemek

Ardından Directories menüsüne MPICH kütüphaneleri eklenir. Bunun için sırasıyla Library Directories ve Include Directories bölümlerine C:\Program Files\MPICH2\lib ve C:\Program Files\MPICH2\include klasörleri eklenir. Bu adımla beraber derleyici ayarları tamamlanmış olur. Projenin derlenip çalıştırılması için üst menüden TDM-GCC 32 Bit Release derleyicisi seçilir. Ardından F9 tuşu ile derleme işlemi başlatılır. Derleme tamamlandıktan sonra geçerli klasörde aşağıdaki gibi bir görüntü elde edilecektir. (Şekil 3.28)

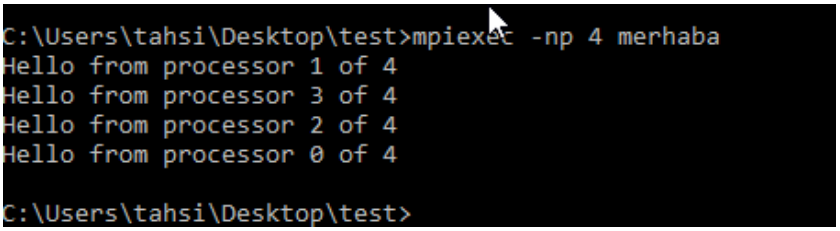


Şekil 3.28 Derleme İşlemi

Burada merhaba.exe paralel programımız olmaktadır. Programımızı test etmek için proje klasöründe bir konsol penceresi açılır. Ardından

```
1 mpiexec -np 4 merhaba
```

komutu ile program çalıştırılır. Burada -np programın sadece kendi makinemizin çekirdeklerinde çalışacağını belirtmektedir. 4 sayısı ise kaç çekirdek ile çalışacağını belirtmektedir. Eğer bir hatayla karşılaşılmazsa aşağıdaki gibi bir ekran görüntüsü karşınıza gelecektir. (Şekil 3.29)



Şekil 3.29 Programın Çıktısı

3.6.2 Linux

Bu bölümde örnek bir uygulamanın Linux işletim sisteminde çalıştırılması anlatılmıştır. Öncelikle proje için mpi_ornek.c dosyası oluşturulur ve içerisine

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

int main(int argc, char** argv) {
MPI_Init(NULL, NULL);
printf("Merhaba");
MPI_Finalize();
}
```

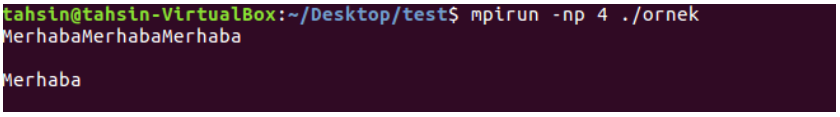
kodları yazılır. Bu kodların açıklamaları ileriki bölümlerde verilmiştir. Örneğin yazılmış olduğu klasörde

```
mpicc -o ornek mpi_ornek.c
```

komutu ile örnek derlenir.

```
mpirun -np 4 ./ornek
```

komutu ile programı çalıştırılır. (Şekil 3.30)



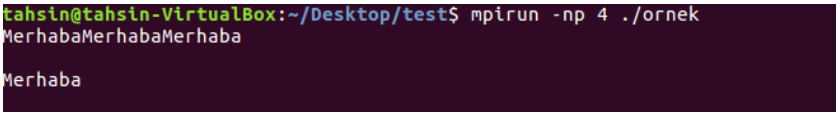
Şekil 3.30 Linux'ta Programın Çıktısı

3.6.3 MacOS

Bu bölümde örnek bir uygulamanın macOS işletim sisteminde çalıştırılması anlatılmıştır.

```
1 mpiexec -np 4 programIsmi
```

komutu ile MPI program MacOS üzerinde çalıştırılabilir. (Şekil 3.31)



Şekil 3.31 MacOS'ta Programın Çıktısı

4. MPICH FONKSİYONLARI

4.1 Temel Fonksiyonlar

Tüm MPI programlarında programın paralel olarak çalışması için gerekli olan bazı temel fonksiyonlar vardır. Bu bölümde bu fonksiyonların detayları anlatılmıştır.

4.1.1 MPI_Init

Bu fonksiyon MPI işlemlerini başlatır. Paralel programlamanın başlayacağı bölümlerde MPI bu fonksiyon ile başlatılır. Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Init(int *argc, char **argv)
```

4.1.2 MPI_Comm_Rank

Bu fonksiyon programın o anda çalıştırıldığı işlemcinin sırasını vermektedir. Genellikle işlemciler sıfırdan başlanarak sıralanmaktadır. Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Bu fonskiyonda MPI_COMM_WORLD kullanılarak tüm işlemcilerin kümesi seçilebilir.

4.1.3 MPI_Comm_Size

Bu fonksiyon verilen kümedeki işlemcilerin sayısını vermektedir. Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Comm_size(MPI_Comm comm, int *size)
```

Bu fonksiyonda MPI_COMM_WORLD kullanılarak tüm işlemcilerin kümesi seçilebilir.

4.1.4 MPI_Finalize

Bu fonksiyon tüm MPI işlemlerini program için sonlandırır. Bu fonksiyondan sonra yazılan kodlar paralel olarak çalışmazlar. Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Finalize()
```

4.2 Noktadan Noktaya İletişim

4.2.1 MPI_Send

Bu fonksiyon istenilen verinin bir işlemciye gönderilmesi sırasında kullanılır. Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
2 int tag, MPI_Comm comm)
```

MPI_Send fonksiyonuna benzer şekilde çalışan farklı fonksiyonlar vardır. Bu fonksiyonların incelenmesinden önce bloklanmış (blocking) ve bloklanmamış (non-blocking) iletişimin ne olduğu incelenmiştir.

Bloklanmış iletişimde veri gönderilir veya alınırken işlem tamamlanmadan diğer bir işleme geçilmez. Bloklanmamış iletişimde ise veri gönderimi veya alımı cevap beklenmeden yapılır. Buna göre MPI_Send fonksiyonu benzerleri ve detayları aşağıdaki şekildedir (Çizelge 4.1)

Çizelge 4.1 MPI_Send Fonksiyonları

MPI_Ssend	Bloklanmış
MPI_Isend	Bloklanmamış
MPI_Send	Bloklanmış

4.2.2 MPI_Recv

Bu fonksiyon işlemciden gönderilen verinin alınması için kullanılır. Fonksiyonun kullanımı aşağıdaki şekildedir.

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status *status)
```

Uygulama 4.1

1'den 1000'e kadar olan sayıların toplamlarının karesi ve karelerinin toplamının farkını bulalım. Bunun için iki işlemci kullanılmıştır ve 1'den 1000'e kadar olan sayıların bulunduğu dizi ikiye bölünerek bu işlemcilere gönderilmiştir. Bu işlemcilerde hesaplanan değer toplanarak gerekli sonuca ulaşılmıştır.

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

#define AZAMI 100

//Bu fonksiyon sadece süreyi uzatmak için kullanılıyor
int sureyi_uzat(){
int c = 1, d = 1;
for ( c = 1 ; c <= 100 ; c++ )
for ( d = 1 ; d <= 400 ; d++ )
{}
return 0;
}

main(int argc, char **argv)
{
//İşlemlerde kullanılacak dizi
unsigned long dizi[AZAMI],toplam,ara_toplam1 =0,ara_toplam2 =0;
//Toplam işlemci sayısı ve o anda işlemi yapan
//işlemciyi belirten değişkenler
int my_id, num_procs,init,k;
//MPI_Recv fonksiyonu için kullanılan durum değişkeni
MPI_Status status;

//MPI başlatılıyor.
init =MPI_Init(&argc, &argv);
//my_id ve num_procs değişkenlerine gerekli değerler atanıyor
MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
```

```
32 MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
33
34 //Eğer işlemci sayısı ikiye eşit değilse
35 //program sonlandırılıyor
36 if(num_procs!=2){
37 printf("Islemci sayısı iki olmalı!");
38 MPI_Abort(MPI_COMM_WORLD,init);
39 exit(0);
40 }
41
42 //0 numaralı işlemciyi görevleri dağıtmak için ve seri programlama
43 //için geçen süreyi hesaplamada kullanacağız.
44 if( my_id == 0 ) {
45 //Geçen süreyi hesaplamak için kullanacağımız değişkenler.
46 clock_t s,d;
47 //Diziye değişkenler atanıyor.
48 for ( k =1 ; k <= AZAMI; k++ ) dizi[k] = k;
49
50 /*
51 * SERİ PROGRAMLAMA BAŞLANGIÇ
52 */
53
54 //Süre başlatıldı.
55 s =clock();
56 //Karelerinin toplamı hesaplanıyor
57 for ( k =1 ; k <= AZAMI; k++ ){
58 ara_toplam1 =ara_toplam1+pow(dizi[k],2);
59 sureyi_uzat();
60 }
61 //Toplamların karesi hesaplanıyor
62 for ( k =1 ; k <= AZAMI; k++ ){
63 ara_toplam2 =ara_toplam2+dizi[k];
64 sureyi_uzat();
65 }
66 ara_toplam2 =ara_toplam2*ara_toplam2;
67 //Çıkan değerlerin farkı hesaplanıyor
68 toplam =ara_toplam2-ara_toplam1;
69 //Süre durduruldu
70 d =clock();
71 //Seri programlama için geçen süre ekrana yazdırılıyor
72 printf("Seri => Gecen sure : %f milisaniye \n",difftime(d,s));
73
74 /*
75 * SERİ PROGRAMLAMA BİTİŞ
76 */
77
78 //Paralel programlamada geçen süreyi hesaplama için
79 //değişken oluşturuldu
80 //ve süre başlatıldı
81 clock_t say;
82 say =clock();
83
84 //Toplam tekrar sıfıra eşitlendi
85 toplam =0;
86 ara_toplam1 =0;
87
```

```
88 //1 numaralı işlemciye dizi gönderiliyor
89 MPI_Send(&dizi,AZAMI,MPI_INT,1,123,MPI_COMM_WORLD);
90
91 //0 numaralı işlemcinin payına düşen hesaplama yapılıyor
92 for ( k =1 ; k <= AZAMI; k++ ){
93 ara_toplam1 =ara_toplam1+pow(dizi[k],2);
94 sureyi_uzat();
95 }
96 //1 numaralı işlemciden hesaplanan toplam alınıyor
97 MPI_Recv(&ara_toplam2, 1, MPI_INT, 1, 1234,MPI_COMM_WORLD,&status);
98
99 //İki işlemciden gelen toplamlar birleştiriliyor
100 toplam =ara_toplam2-ara_toplam1;
101
102
103 clock_t dur;
104 dur =clock();
105
106 printf("Paralel => Gecen sure : %f milisaniye \n",difftime(dur,say));
107 }else{
108 ara_toplam2 =0;
109 MPI_Recv(&dizi,AZAMI,MPI_INT,0,123,MPI_COMM_WORLD,&status);
110 for ( k =1 ; k <= AZAMI; k++ ){
111 ara_toplam2 =ara_toplam2+dizi[k];
112 sureyi_uzat();
113 }
114 ara_toplam2 =ara_toplam2*ara_toplam2;
115 MPI_Send(&ara_toplam2,1,MPI_INT,0,1234,MPI_COMM_WORLD);
116 }
117 //MPI sonlandırılıyor
118 MPI_Finalize();
119 }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 4.1)

```
Seri => Gecen  sure :  16.000000 milisaniye
Paralel => Gecen  sure :  15.000000 milisaniye
```

Şekil 4.1 Uygulama 4.1 Sonucu

Uygulama 4.2

1’den 10’a kadar olan sayılardan 3 veya 5’e bölünenlerin 3, 5, 6, 9 toplamı 23tür. 1’den 1000’e kadar olan sayılardan 3 veya 5’e bölünenlerin toplamını bulalım. Bunun için iki işlemci kullanılmıştır ve 1’den 1000’e kadar olan sayıların bulunduğu dizi ikiye bölünerek bu işlemcilere gönderilmiştir. Bu işlemcilerde 3 veya 5’e bölünen sayılar ara toplama eklenerek

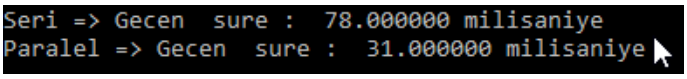
ana işlemciye gönderilmiştir. Ara toplamlar ise bir numaralı işlemcide toplanarak sonuca ulaşılmıştır.

```
1 #include <mpi.h>
2 #include <time.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 #define AZAMI 1000
7
8 //Bu fonksiyon sadece süreyi uzatmak için kullanılıyor
9 int sureyi_uzat(){
10 int c = 1, d = 1;
11 for ( c = 1 ; c <= 100 ; c++ )
12 for ( d = 1 ; d <= 400 ; d++ )
13 {}
14 return 0;
15 }
16
17 main(int argc, char **argv)
18 {
19 //İşlemlerde kullanılacak dizi
20 unsigned long dizi[AZAMI],toplam,ara_toplam =0;
21 //Toplam işlemci sayısı ve o anda işlemi yapan işlemciyi
22 //belirten değişkenler
23 int my_id, num_procs,init,k;
24 //MPI_Recv fonksiyonu için kullanılan durum değişkeni
25 MPI_Status status;
26
27 //MPI başlatılıyor.
28 init =MPI_Init(&argc, &argv);
29 //my_id ve num_procs değişkenlerine gerekli değerler atanıyor
30 MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
31 MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
32
33 //Eğer işlemci sayısı üçten büyük veya eşit değilse
34 //program sonlandırılıyor
35 if(num_procs!=2){
36 printf("Islemci sayısı iki olmalı!");
37 MPI_Abort(MPI_COMM_WORLD,init);
38 exit(0);
39 }
40
41 //0 numaralı işlemciyi görevleri dağıtmak için ve seri programlama
42 //için geçen süreyi hesaplamada kullanacağız.
43 if( my_id == 0 ) {
44 //Geçen süreyi hesaplamak için kullanacağımız değişkenler.
45 clock_t s,d;
46 //Diziye değişkenler atanıyor.
47 for ( k =1 ; k <= AZAMI-1; k++ ) dizi[k] = k;
48
49 /*
50 * SERİ PROGRAMLAMA BAŞLANGIÇ
```

```
51 */
52
53 //Süre başlatıldı.
54 s =clock();
55 //Dizideki 3'e veya 5'e bölünen elemanların toplamının
56 //tutulduğu değişken
57 toplam = 0;
58 //Tüm dizi elemanlarının 3'e veya 5'e bölünüp bölünmediği
59 //kontrol ediliyor
60 for ( k =1 ; k <= AZAMI-1; k++ ){
61 //Eğer sıradaki eleman 3'e veya 5'e bölünüyorsa toplam
62 //değişkenine ekleniyor
63 if(dizi[k]%3 ==0 || dizi[k]%5 ==0) toplam = toplam + dizi[k];
64 sureyi_uzat();
65 }
66 //Süre durduruldu
67 d =clock();
68 //Seri programlama için geçen süre ekrana yazdırılıyor
69 printf("Seri => Gecen sure : %f milisaniye \n",difftime(d,s));
70
71 /*
72 * SERİ PROGRAMLAMA BİTİŞ
73 */
74
75 //Paralel programlamada geçen süreyi hesaplama için değişken
76 //oluşturuldu ve süre başlatıldı
77 clock_t say;
78 say =clock();
79
80 //Toplam tekrar sıfıra eşitlendi
81 toplam =0;
82
83 //1 numaralı işlemciye dizi gönderiliyor
84 MPI_Send(&dizi,AZAMI,MPI_INT,1,123,MPI_COMM_WORLD);
85
86 //0 numaralı işlemcinin payına düşen hesaplama yapılıyor
87 for ( k =1 ; k <= (AZAMI / 2); k++ ) {
88 if(dizi[k]%3 ==0 || dizi[k]%5 ==0) toplam = toplam + dizi[k];
89 sureyi_uzat();
90 }
91
92 //1 numaralı işlemciden hesaplanan toplam alınıyor
93 MPI_Recv(&ara_toplam, 1, MPI_INT, 1, 1234,MPI_COMM_WORLD,&status);
94
95 //İki işlemciden gelen toplamlar birleştiriliyor
96 toplam =toplam+ara_toplam;
97
98 clock_t dur;
99 dur =clock();
100
101 printf("Paralel => Gecen sure : %f milisaniye \n",difftime(dur,say));
102 }else{
103 MPI_Recv(&dizi,AZAMI,MPI_INT,0,123,MPI_COMM_WORLD,&status);
104 for ( k =((AZAMI/2)+1) ; k <= AZAMI; k++ ) {
105 if(dizi[k]%3 ==0 || dizi[k]%5 ==0) ara_toplam = ara_toplam + dizi[k];
106 sureyi_uzat();
```

```
107 }
108 MPI_Send(&ara_toplam,1,MPI_INT,0,1234,MPI_COMM_WORLD);
109 }
110 //MPI sonlandırılıyor
111 MPI_Finalize();
112 }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 4.2)



Şekil 4.2 Uygulama 4.2 Sonucu

Uygulama 4.3

Programın çalıştığı anda rastgele üretilen bir dizinin elemanlarını toplayan paralel programı yazalım. Bunun için dizi elemanları işlemci sayısına göre parçalara bölünür ve bu parçalar işlemcilere gönderilir. İşlemciler alınan parçalarda toplama işlemini yaptıktan sonra bir numaralı işlemciye sonuçları geri döndürürler. Bu sonuçlar birleştirilerek dizinin elemanlarının toplamına ulaşılır.

```
1  #include <mpi.h>
2  #include <time.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <math.h>
6  #include <string.h>
7
8  //Dizinin eleman sayısını belirten sabit
9  #define TOPLAM_DIZI 100000
10
11 #define GELEN_MESAJ 2
12 #define GIDEN_MESAJ 1
13
14 //İşlem süresini uzatmak için kullanılan fonksiyon
15 double cevir(int j){
16 double ara = sin(sqrt(j)) / 0.123456 * cos(sqrt(j));
17 ara = ara * 3,14 / log(j);
18 ara =j*2;
19 return ara;
20 }
21
22 //İki sayıyı birleştirmek için kullandığımız fonksiyon
```

```
23 int concat(int x, int y){
24 char str1[20];
25 char str2[20];
26 sprintf(str1,"%d",x);
27 sprintf(str2,"%d",y);
28 strcat(str1,str2);
29 return atoi(str1);
30 }
31
32 main(int argc, char **argv)
33 {
34 //İşlemlerde kullanılacak dizi
35 unsigned long dizi[TOPLAM_DIZI];
36 //Toplam işlemci sayısı ve o anda işlemi yapan işlemciyi
37 //belirten değişkenler
38 int my_id, num_procs,init,isciler =1,ortalama_sayi,kalan,sayi_gonder,k;
39 //MPI_Recv fonksiyonu için kullanılan durum değişkeni
40 MPI_Status status;
41
42 //MPI başlatılıyor.
43 init =MPI_Init(&argc, &argv);
44 //my_id ve num_procs değişkenlerine gerekli değerler atanıyor
45 MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
46 MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
47
48 //Eğer işlemci sayısı üçten büyük veya eşit değilse
49 //program sonlandırılıyor
50 if(num_procs<3){
51 printf("Islemci sayısı olarak üçten büyük veya eşit bir sayı giriniz!");
52 MPI_Abort(MPI_COMM_WORLD,init);
53 exit(1);
54 }
55
56 //0 numaralı işlemciyi görevleri dağıtmak için ve seri programlama
57 //için geçen süreyi hesaplamada kullanacağız.
58 if( my_id == 0 ) {
59 //Geçen süreyi hesaplamak için kullanacağımız değişkenler.
60 clock_t s,d;
61 //Diziye değişkenler atanıyor.
62 srand(time(0));
63 for ( k =0 ; k <= TOPLAM_DIZI-1; k++ ) dizi[k] = rand();
64
65 /*
66 * SERİ PROGRAMLAMA BAŞLANGIÇ
67 */
68
69 //Süre başlatıldı.
70 s =clock();
71 //Dizideki elemanların toplamının tutulduğu değişken
72 unsigned long tp = 0;
73 //Tüm dizi elemanları sırayla cevir fonksiyonundan geçirip
74 //tp değişkenine ekleniyor.
75 for ( k =0 ; k <= TOPLAM_DIZI-1; k++ ) tp = tp + round(cevir(dizi[k]));
76 //Seri toplamın değeri ekrana yazdırılıyor.
77 printf("Seri Toplam = %ld \n",tp);
78 //Süre durduruldu
```

```
79 d =clock();
80 //Seri programlama için geçen süre ekrana yazdırılıyor
81 printf("Seri => Gecen sure : %f milisaniye \n",difftime(d,s));
82
83 /*
84 * SERİ PROGRAMLAMA BİTİŞ
85 */
86
87 //Paralel programlamada geçen süreyi hesaplama için
88 //değişken oluşturuldu ve süre başlatıldı
89 clock_t say;
90 say =clock();
91
92 //İşlemci başına ortalama gönderilecek dizi elemanı
93 ortalama_sayi =TOPLAM_DIZI/(num_procs-1);
94 //Eğer dizi elamanları işlemcilere tam bölünemiyorsa kalan hesaplanıp
95 //son işlemciye gönderilecek
96 kalan =TOPLAM_DIZI%(num_procs-1);
97 //İlk elemandan başlanıyor
98 sayi_gonder =0;
99
100 //Gönderilecek işlemci sayısı num_procs-1
101 //Dizi elemanları işlemcilere gönderiliyor
102 for(isciler =1;isciler<=num_procs-1;isciler++){
103 //Eğer son işlemciye gönderiliyorsa kalan eleman varsa
104 //onları da gönderiyoruz
105 if(isciler ==num_procs-1){
106 ortalama_sayi =ortalama_sayi+kalan;
107 }
108 //Eleman aralıkları gönderiliyor
109 MPI_Send(&sayi_gonder,1, MPI_INT,isciler,concat(isciler,32),
110 MPI_COMM_WORLD);
111 MPI_Send(&ortalama_sayi,1, MPI_INT,isciler,concat(isciler,42),
112 MPI_COMM_WORLD);
113 //Dizi gönderiliyor
114 MPI_Send(&dizi,TOPLAM_DIZI, MPI_INT,isciler,concat(isciler,52),
115 MPI_COMM_WORLD);
116 //Gönderilen eleman kadar ilk aralığa ekleniyor
117 sayi_gonder =sayi_gonder+ortalama_sayi;
118 }
119
120 //Genel toplam sıfırlandı
121 tp =0;
122 unsigned long ara_toplam = 0;
123
124 //Toplamlar işlemcilerden alınıyor
125 for(isciler =1;isciler<=num_procs-1;isciler++){
126 MPI_Recv(&ara_toplam,1,MPI_INT,isciler,concat(isciler,62)
127 , MPI_COMM_WORLD, &status);
128 tp =tp+ara_toplam;
129 }
130
131 //Paralel toplam yazdırılıyor
132 printf("Paralel Toplam = %ld \n",tp);
133
134 //Süre durdurulup ekrana yazdırılıyor
```

```
clock_t dur;
dur =clock();
printf("Paralel => Gecen sure : %f milisaniye \n",difftime(dur,say));
}else{
//0 numaralı işlemciden gelen veriler alınıyor
MPI_Recv(&sayi_gonder,1,MPI_INT,0,concat(my_id,32),
MPI_COMM_WORLD, &status);
MPI_Recv(&ortalama_sayi,1,MPI_INT,0,concat(my_id,42),
MPI_COMM_WORLD, &status);
MPI_Recv(&dizi,TOPLAM_DIZI,MPI_INT,0,concat(my_id,52),
MPI_COMM_WORLD, &status);
//Gelen verilere göre toplam hesaplanıyor
unsigned long isci_toplami = 0;
for ( k =sayi_gonder ; k <= (sayi_gonder+ortalama_sayi-1); k++ ) {
isci_toplami = isci_toplami + round(cevir(dizi[k]));
}
//Toplam 0 numaralı işlemciye geri gönderiliyor
MPI_Send(&isci_toplami,1,MPI_INT,0,concat(my_id,62), MPI_COMM_WORLD);
}
//MPI sonlandırılıyor
MPI_Finalize();
}
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 4.3)

```
Seri Toplam = -1016103120
Seri => Gecen  sure :  16.000000 milisaniye
Paralel Toplam = -1016103120
Paralel => Gecen  sure :  15.000000 milisaniye
```

Şekil 4.3 Uygulama 4.3 Sonucu

Uygulama 4.4

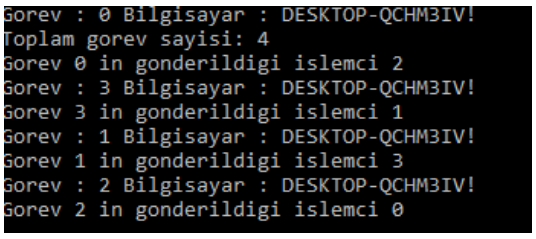
MPI_Irecv ve MPI_Isend kullanarak bloklanmamış iletişim yapan bir program yazınız. Bunun için öncelikle uygulamanın çalıştırıldığı işlemciler birbirleriyle eşleştiriliyor. Belirlenen mesajların eşleştirilen işlemciler tarafından birbirine gönderimi sağlanıyor. Gelen mesaj ekrana yazdırılıyor.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define MASTER 0

```

```
6  int main (int argc, char *argv[])
7  {
8  int numtasks, taskid, len;
9  char hostname[MPI_MAX_PROCESSOR_NAME];
10 int partner, message;
11 MPI_Status stats[2];
12 MPI_Request reqs[2];
13
14 MPI_Init(&argc, &argv);
15 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
16 MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
17 MPI_Get_processor_name(hostname, &len);
18 //İşlemi yapanın numarası ve bilgisayar ismi yazdırılıyor
19 printf ("Gorev : %d Bilgisayar : %s!\n", taskid, hostname);
20 //0 numaralı kaç tane görev olduğunu yazdırıyor
21 if (taskid == MASTER)
22 printf("Toplam gorev sayisi: %d\n",numtasks);
23
24 /* Gönderme ve almada kullanılacak partnerler seçiliyor */
25 if (taskid < numtasks/2)
26 partner = numtasks/2 + taskid;
27 else if (taskid >= numtasks/2)
28 partner = taskid - numtasks/2;
29
30 //Mesaj gönderme ve alma işlemi başladı
31 MPI_Irecv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &reqs[0]);
32 MPI_Isend(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &reqs[1]);
33
34 /* İstekler tamamlanana kadar duraklat */
35 MPI_Waitall(2, reqs, stats);
36
37 printf("Gorev %d in gonderildigi islemci %d\n",taskid,message);
38
39 MPI_Finalize();
40
41 }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 4.4)



Şekil 4.4 Uygulama 4 Sonucu

4.3 Toplu İletişim

Bu bölüme kadar bahsedilen fonksiyonlar verileri noktadan noktaya iletebiliyordu. Mesela aynı veriyi diğer işlemcilere göndermek için send fonksiyonunun defalarca kullanılması gerekiyordu. Daha kullanışlı bir metod olarak bu tarz işlemlerde MPI'ın toplu iletişim fonksiyonları kullanılabilir. Bu sayede;

- Noktadan noktaya iletişime oranla yazılan kod azalır.
- Toplu iletişimde elimizdeki işlemci kümesindeki tüm işlemciler iletişime dahil edilir.(MPI_COMM_WORLD) Program yazılırken bunun yerine farklı bir küme tanımlayarak o da dahil edilebilir.
- Toplu iletişimde bir işlemcinin iletişime dahil olmamasından dolayı 'program çökebilir, çıktı olarak alınan sonuç hatalı olabilir,...'' gibi birçok hata yaşanabilir.

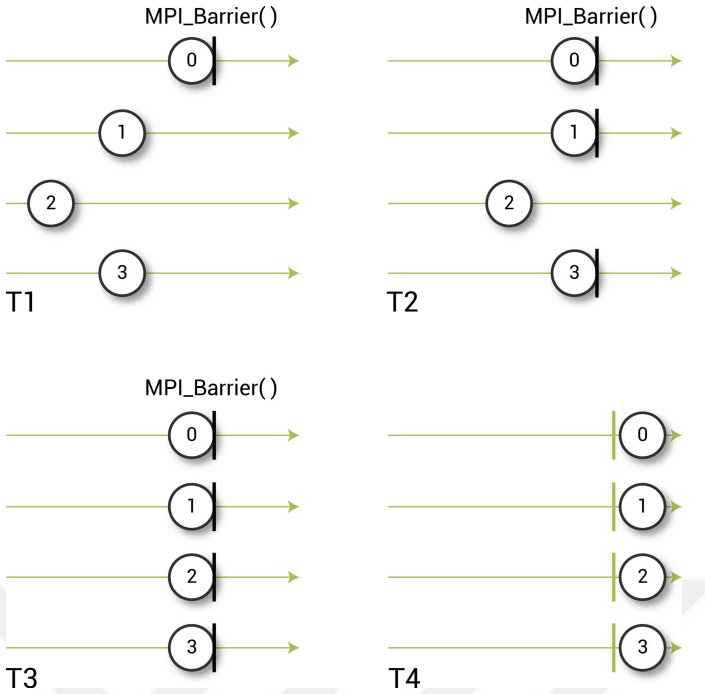
Bu bölümde toplu iletişimde kullanılacak fonksiyonlar incelenmiştir.

4.3.1 MPI_Barrier

Bu fonksiyon, kullanıldığı noktada işlemlerin senkronize olmasını sağlar. T1 grafiğinde sıfır numaralı işlemci MPI_Barrier fonksiyonunu çağırır. Sıfır numaralı işlemci beklemedeyken T2 grafiğinde bir ve üç numaralı işlemciler de MPI_Barrier fonksiyonunu çağırır. T2 grafiğinde bütün işlemciler MPI_Barrier fonksiyonunu çağırdıktan sonra T4 grafiğinde senkronize olarak işleme devam ederler. (Şekil 4.5)

Fonksiyonun kullanımı aşağıdaki şekildedir.

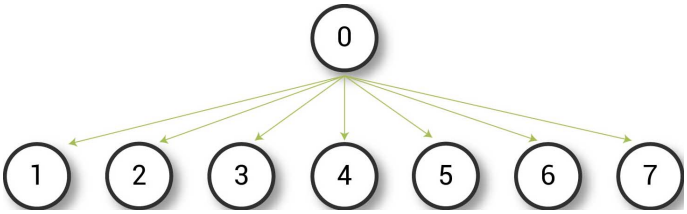
```
1 MPI_Barrier( MPI_Comm comm )
```



Şekil 4.5 MPI_Barrier

4.3.2 MPI_Bcast

Bu fonksiyon seçilen ana işlemciden diğer tüm işlemcilere mesajı gönderir. (Şekil 4.6)



Şekil 4.6 MPI_Bcast

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
2  MPI_Comm comm )
```

Uygulama 4.5

MPI_Bcast kullanarak herhangi bir mesajı tüm işlemcilere gönderen paralel programı yazalım. Bunun için ilk işlemcide mesaj belirlenir. Ardından bu mesaj tüm işlemcilere gönderilir. Diğer işlemciler tarafından alınan mesaj ekrana yazdırılır.

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char **argv) {
5
6  //Kullanılacak değişkenler
7  int my_id,num_procs,mesaj;
8
9  //Paralel programlama başladı
10 MPI_Init(&argc,&argv);
11 MPI_Comm_size(MPI_COMM_WORLD,&num_procs);
12 MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
13
14 //Ana işlemci mesajı oluşturuyor
15 if(my_id ==0){
16 mesaj =1234;
17 }
18
19 //Mesaj tüm işlemcilere gönderiliyor
20 MPI_Bcast(&mesaj,1,MPI_INT,0,MPI_COMM_WORLD);
21
22 //Gönderilen mesaj yazdırılıyor
23 printf("Gelen mesaj :%d .Islemci numarasi : %d \n",mesaj,my_id);
24
25 //Paralel programlama tamamlandı
26 MPI_Finalize();
27 return 0;
28 }
```

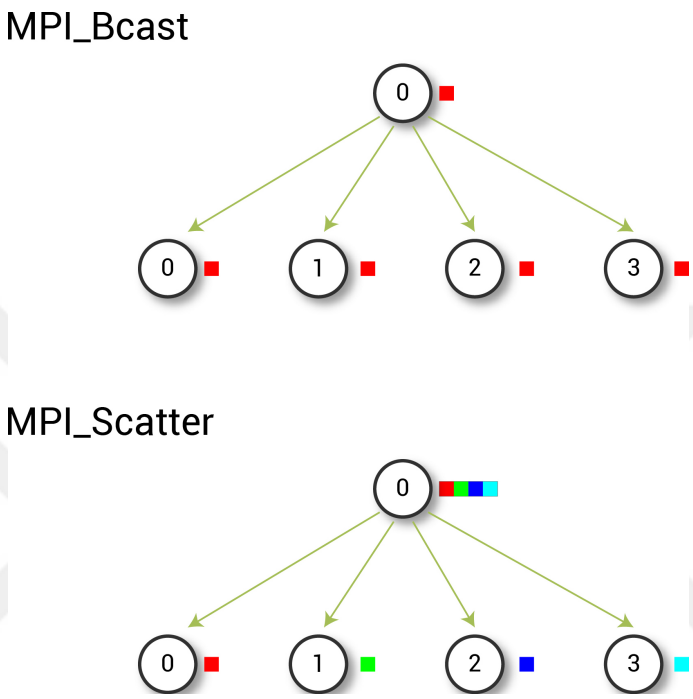
Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 4.7)

```
Gelen mesaj :1234 .Islemci numarasi : 2
Gelen mesaj :1234 .Islemci numarasi : 1
Gelen mesaj :1234 .Islemci numarasi : 0
Gelen mesaj :1234 .Islemci numarasi : 3
```

Şekil 4.7 Uygulama 4.5 Sonucu

4.3.3 MPI_Scatter

MPI_Bcast aldığı veriyi tüm işlemcilere gönderir. MPI_Scatter ise MPI_Bcast'in aksine veriyi parçalara ayırıp diğer işlemcilere gönderir. Örneğin, [1,2,3,4] dizisini [1],[2],[3],[4] şeklinde parçalara ayırarak dört ayrı işlemciye gönderir. (Şekil 4.8)



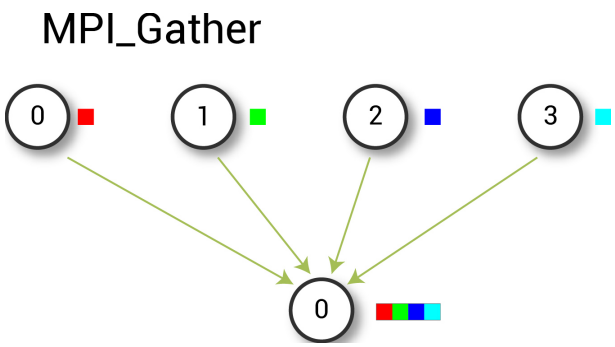
Şekil 4.8 MPI_Scatter

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
2 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
3 MPI_Comm comm)
```

4.3.4 MPI_Gather

Bu fonksiyon diğer işlemcilerden verileri toplayıp ana işlemciye gönderir. Gönderilen verilerin üzerinde işlemler tamamlandıktan sonra bir numaralı işlemcide MPI_Gather ile toplanarak paralel programın sonuçları işlenir. (Şekil 4.9)



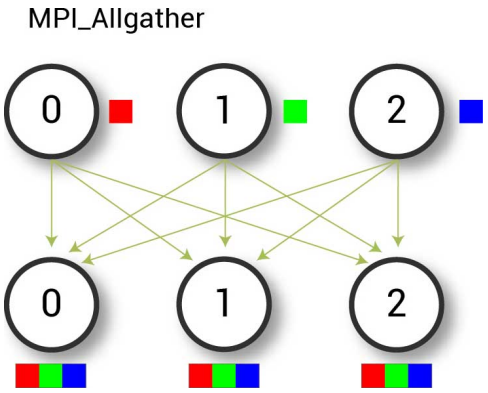
Şekil 4.9 MPI_Gather

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
2 void *recvbuf,int recvcount, MPI_Datatype recvtype, int root,
3 MPI_Comm comm)
```

4.3.5 MPI_Allgather

MPI_Gather'da işlemcilerden gönderilen veriler bir işlemcide toplanarak programa devam ediliyor. MPI_Allgather'da ise işlemcilerden gönderilen veriler toplanarak tüm işlemcilere gönderilir. (Şekil 4.10)



Şekil 4.10 MPI_Allgather

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

Uygulama 4.6

MPI_Scatter ve MPI_Gather kullanarak rastgele üretilen bir dizinin ortalamasını bulan paralel programı yazalım. Bunun için bu uygulamada işlemci başına düşecek eleman sayısı 100 seçilmiştir. MPI_Scatter yardımıyla uygulamayı çalıştıran işlemcilere bu elemanlar yüzer yüzer gönderilir. MPI_Gather yardımıyla da ortalaması bulunan dizi parçalarının sonuçları işlemcilerden toplanır. Bu parçalar toplanarak dizinin ortalaması bulunur.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <assert.h>
#include <time.h>
#include <math.h>

// 0 ile 1 arasında rastgele sayı üreten fonksiyon
float *create_rand_nums(int num_elements) {
float *rand_nums;
//Rastgele değerler üretilip geri döndürülüyor
int i;
for (i = 0; i < num_elements; i++) {
rand_nums[i] = (rand() / (float)RAND_MAX);
}
return rand_nums;
}

// Dizinin ortalamasını hesaplayan fonksiyon
float compute_avg(float *array, int num_elements) {
float sum = 0.f;
int i;
for (i = 0; i < num_elements; i++) {
sum + = array[i];
}
return sum / num_elements;
}

int main(int argc, char** argv) {

//İşlemci başına düşecek eleman sayısı
int num_elements_per_proc = 100;

//Rastgele değerler için kullanılıyor
srand(time(NULL));
```

```
36
37 //Paralel programlama başladı
38 MPI_Init(&argc, &argv);
39
40 //Toplam işlemci sayısı ve şu an kullanılan işlemciyi
41 //belirten değişkenler
42 int world_rank,world_size;
43 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
44 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
45
46 //0 numaralı işlemcide rastgele sayıların olduğu dizi üretilecek
47 float *rand_nums = NULL;
48 if (world_rank == 0) {
49 //Rastgele sayılar
50 rand_nums = create_rand_nums(num_elements_per_proc * world_size);
51 }
52
53 //Dizi parçaları
54 float *sub_rand_nums;
55
56 // Parçalar 100er 100er 0 numaralı işlemciden diğerlerine gönderiliyor
57 MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums,
58 num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);
59
60 // Gönderilen parçalar için hesaplama yapılıyor
61 float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);
62
63 // Toplamlar 0 numaralı işlemcide toplanıyor
64 float *sub_avgs = NULL;
65 MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
66 MPI_COMM_WORLD);
67
68 //0 numaralı işlemci alınan değerlerin toplamını yazdırıyor
69 if (world_rank == 0) {
70 float avg = compute_avg(sub_avgs, world_size);
71 printf("Paralelden Gelen Deger : %f\n", avg);
72 //Seri programlama ile hesaplandı
73 float original_data_avg =compute_avg(rand_nums,
74 num_elements_per_proc * world_size);
75 printf("Seriden Gelen Deger : %f\n", original_data_avg);
76 }
77
78 //Tüm işlemciler MPI_Barrier fonksiyonunu çalıştırdıktan sonra
79 //MPI_Finalize çalıştırılabilir
80 MPI_Barrier(MPI_COMM_WORLD);
81 MPI_Finalize();
82 }
```

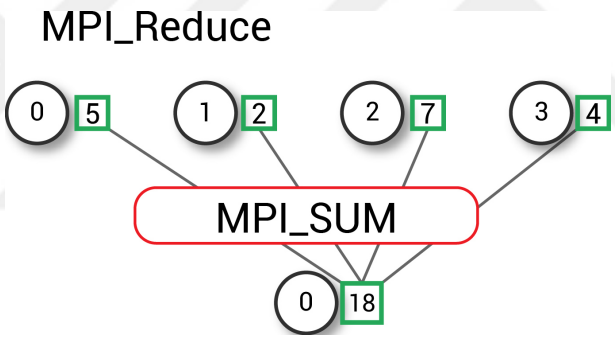
Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 4.11)

```
1: DESKTOP-QCHM3IV: 123
2: DESKTOP-QCHM3IV: 123
3: DESKTOP-QCHM3IV: 123
```

Şekil 4.11 Uygulama 4.6 Sonucu

4.3.6 MPI_Reduce

Bu fonksiyon, seçilen işlemi toplanan verilere uygular. Uygulama 4.6'daki dizi ortalaması işlemini ele alırsak, işlemcilerden toplanan veriler MPI_Reduce fonksiyonunda toplama işlemine sokularak dizinin ortalaması ayrı bir toplama işlemine ihtiyaç olmadan bulunabilir. (Şekil 4.12)



Şekil 4.12 MPI_Reduce

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
2 MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

Bu fonksiyonda kullanılabilecek işlemler ise şu şekildedir: (Çizelge 4.2)

Çizelge 4.2 MPI_Reduce Fonksiyonları

MPI_MAX	Maksimum
MPI_MIN	Minimum
MPI_SUM	Toplam
MPI_PROD	Çarpım
MPI_LAND	Mantıksal AND
MPI_BAND	Bit-Wise AND
MPI_LOR	Mantıksal OR
MPI_BOR	Bit-Wise OR
MPI_LXOR	Mantıksal XOR
MPI_BXOR	Bit-Wise XOR
MPI_MAXLOC	Maksimum değer ve yeri
MPI_MINLOC	Minimum değer ve yeri

4.3.7 MPI_Allreduce

Bu fonksiyon MPI_Reduce'un aksine parametre olarak ana işlemciyi almaz. Bunun yerine veriler bütün işlemcilere dağılır. Yani MPI_Reduce'un ardından MPI_Bcast uygulanır.

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Allreduce(const void *sendbuf, void *recvbuf,
2 int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

4.3.8 MPI_Reduce_scatter

Bu fonksiyon önce verilere MPI_Reduce fonksiyonunu uygular ardından çıkan veriyi MPI_Scatter ile diğer işlemcilere dağıtır.

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Reduce_scatter(const void *sendbuf, void *recvbuf,
2 const int recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Uygulama 4.7

MPI_Reduce kullanarak rastgele üretilen dizinin elemanlarının toplamını ve ortalamasını bulan paralel program yazalım. Bunun için rastgele oluşturulan dizi tüm işlemcilere gönderilir. Ardından MPI_Reduce fonksiyonu kullanılarak bu gelen değerler toplaranak sonuca ulaşılır.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include <assert.h>
5
6 // 0 ile 1 arasında rastgele sayı üreten fonksiyon
7 float *create_rand_nums(int num_elements) {
8 float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
9 assert(rand_nums != NULL);
10 int i;
11 for (i = 0; i < num_elements; i++) {
12 rand_nums[i] = (rand() / (float)RAND_MAX);
13 }
14 return rand_nums;
15 }
16
17 int main(int argc, char** argv) {
18
19 //İşlemci başına düşecek eleman sayısı
20 int num_elements_per_proc = 100;
21
22 //Rastgele değerler için kullanılıyor
23 srand(time(NULL));
24
25 //Paralel programlama başladı
26 MPI_Init(&argc, &argv);
27
28 //Toplam işlemci sayısı ve şu an kullanılan işlemciyi
29 //belirten değişkenler
30 int world_rank,world_size;
31 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
32 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
33
34 // Tüm işlemciler için rastgele elemanları olan dizi oluşturuluyor
35 srand(world_rank);
36 float *rand_nums = NULL;
37 rand_nums = create_rand_nums(num_elements_per_proc);
38
39 // Şu an çalışan işlemci için üretilen dizinin elemanları toplanıyor
40 float local_sum = 0;
41 int i;
42 for (i = 0; i < num_elements_per_proc; i++) {
43 local_sum + = rand_nums[i];
44 }
```

```
45
46 // Şu an çalışan işlemci için çıkan sonuç ekrana yazdırılıyor
47 printf("%d numarali islemci icin toplam : %f, Ortalama : %f\n",
48 world_rank, local_sum, local_sum / num_elements_per_proc);
49
50 // Şu an çalışan işlemciden gelen toplamı genel toplama ekliyor
51 float global_sum;
52 MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
53 MPI_COMM_WORLD);
54
55 // 0 numaralı işlemci çıkan sonucu ekrana yazdırıyor
56 if (world_rank == 0) {
57 printf("Genel Toplam = %f, Genel Ortalama = %f\n", global_sum,
58 global_sum / (world_size * num_elements_per_proc));
59 }
60
61 //Paralel programlama sonlandırılıyor
62 MPI_Finalize();
63 }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 4.13)

```
1 numarali islemci icin toplam : 49.481133, Ortalama : 0.494811
3 numarali islemci icin toplam : 48.847309, Ortalama : 0.488473
0 numarali islemci icin toplam : 45.798149, Ortalama : 0.457981
Genel Toplam = 194.290878, Genel Ortalama = 0.485727
2 numarali islemci icin toplam : 50.164291, Ortalama : 0.501643
```

Şekil 4.13 Uygulama 4.7 Sonucu

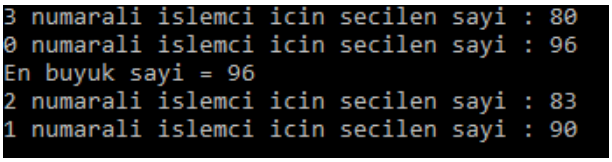
Uygulama 4.8

MPI_Reduce kullanarak rastgele üretilen dizinin en büyük elemanını bulan paralel program yazalım. Bunun için rastgele üretilen dizi, MPI_Reduce fonksiyonuna gönderilir. En büyük değeri bulmak için MPI_MAX seçilmiştir.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include <assert.h>
5
6 int main(int argc, char** argv) {
7
8 //İşlemci başına düşecek eleman sayısı
9 int num_elements_per_proc = 100;
10
```

```
11 //Rastgele değerler için kullanılıyor
12 srand(time(0));
13
14 //Paralel programlama başladı
15 MPI_Init(&argc, &argv);
16
17 //Toplam işlemci sayısı ve şu an kullanılan işlemciyi
18 //belirten değişkenler
19 int world_rank,world_size;
20 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
21 MPI_Comm_size(MPI_COMM_WORLD, &world_size);
22
23 // Tüm işlemciler için rastgele bir eleman seçiliyor
24 int sayi = rand() % 100 + 1;
25
26 // Şu an çalışan işlemci için seçilen sayı ekrana yazdırılıyor
27 printf("%d numarali islemci icin secilen sayi : %i\n",world_rank,sayi);
28
29 // Şu an çalışan işlemciden gelen sayılar arasından en büyüğü seçiliyor
30 int en_buyuk;
31 MPI_Reduce(&sayi, &en_buyuk, 1, MPI_INT, MPI_MAX, 0,MPI_COMM_WORLD);
32
33 // 0 numaralı işlemci çıkan sonucu ekrana yazdırıyor
34 if (world_rank == 0) {
35 printf("En buyuk sayi = %i\n",en_buyuk);
36 }
37
38 //Paralel programlama sonlandırılıyor
39 MPI_Finalize();
40 }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 4.14)



Şekil 4.14 Uygulama 4.8 Sonucu

4.3.9 MPI_Alltoall

MPI_Scatter fonksiyonu işlemcilere gönderilecek olan veriyi parçalara ayırarak işlemcilere gönderiyor. Bu fonksiyon ise iletişim kümesindeki tüm işlemcilere MPI_Scatter fonksiyonunu uygular.

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
2 void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

4.3.10 MPI_Scan

Bu fonksiyon MPI_Reduce fonksiyonunu parça parça uygular.

Fonksiyonun kullanımı aşağıdaki şekildedir.

```
1 MPI_Scan(const void *sendbuf, void *recvbuf, int count,
2 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

5. CLUSTER

Cluster (Bilgisayar Kümesi), paralel programlar veya fazla işlemci gücü isteyen programlar gibi amaçlarla kullanılmak için birden fazla bilgisayarın tek bir bilgisayar gibi davranacak şekilde çalıştırılmasına denir. Genel olarak bu bilgisayarlar bir ağ yardımıyla birbirlerine bağlanır ve aralarındaki iletişim harici bir yazılım tarafından sağlanır (Ataş, Kandıra, 2005).

5.1 Cluster Sınıfları

Clusterlar dört farklı sınıfa sahiptir. Bu sınıflar şunlardır:

1. Izgara Hesaplama Clusterları
2. Yüksek Performanslı Clusterlar
3. Yük Dengeleyici Clusterlar
4. Yüksek Devamlılık Clusterları

Izgara hesaplama clusterlarına BOINC programı örnek gösterilir. Program bilgisayara kurulduktan sonra destek sağlanması istenen projenin seçilmesi istenir. Seçim tamamlandıktan sonra bilgisayar boş kaldığı zamanlarda da bu projenin hesaplamalarını yaparak projeye destek sağlar. Farklı hastalıklar hakkında araştırma yapan Rosetta@home da bu projelerden biridir.

Yüksek performanslı clusterlar genellikle bilimsel hesaplamalarda kullanılır. Beowulf bilgisayar kümeleri buna örnek gösterilebilir (Ngxande, Moorosi, 2014).

Yük dengeleyici clusterlar performansı arttırmak için kullanılırlar. Linux Virtual Server bu sınıfa örnek olarak gösterilebilir (Zhang, 2008).

Yüksek devamlılık clusterları sistem problemleri gibi durumları kendi içinde çözerek yüksek çalışma süresi vermeyi hedeflemektedir. Linux-HA bu sınıfa örnek gösterilebilir (Yeo, 2006).

Oluşturulan bir clusterda işlemci gücü yeterli gelmediğinde yapılmış olan yazılımı bozmadan yeni cluster elemanları eklenebilir. Bu durum clusterların yüksek ölçeklenebilirliğinden kaynaklanmaktadır.

Bu tez çalışmasında Southampton Üniversitesinin kullandığı Raspberry Pi clusterları kullanılmıştır (Cox, 2012).

5.2 Raspberry Pi Nedir?

Raspberry Pi, Raspberry Pi Vakfı tarafından geliştirilen kredi kartı büyüklüğünde bir mini bilgisayardır. (Şekil 5.1)



Şekil 5.1 Raspberry Pi

Raspbian, Ubuntu Mate, Snappy Ubuntu Core, Windows 10 IOT Core, OSMC, LIBREELEC, RISC OS işletim sistemleri Raspberry Pi sitesinden indirilerek kurulumu yapılabilir. Bu çalışmada Raspberry Pi'nin 2 ve 3. modellerinden birer adet bilgisayar kullanılmıştır. SD kartlar hız açısından class 10 olarak seçilmiştir.

Raspberry Pi 3 modelinin özellikleri aşağıdaki gibidir.

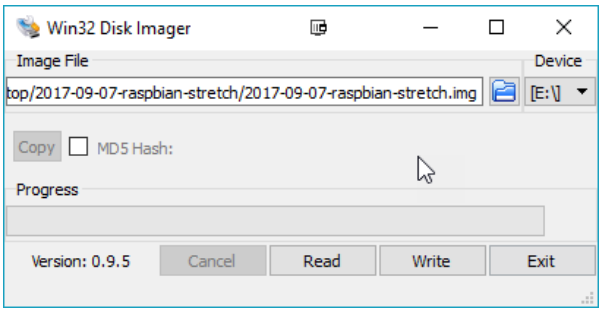
- 1.2GHz 64-bit quad-core ARMv8 işlemci
- 802.11n Wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)

- 1GB RAM
- 4 USB girişi
- 40 GPIO
- HDMI
- Ethernet
- 3.5mm ses girişi
- Micro SD kart yuvası
- VideoCore IV 3D grafik işlemci

5.3 Cluster Oluşturma

5.3.1 İşletim sistemi kurulumu

Bu adım için kullanılmış olan Raspberry Pi'ye uyumlu bir adet SD kart, https://www.raspberrypi.org/downloads/ adresinden indirilecek Raspbian işletim sistemine ve SD karta bu işletim sistemini yazabilecek bir programa (Win32 Disk Imager) ihtiyaç vardır. Bütün dosyalar hazırlandıktan sonra işletim sistemi için indirilmiş olan zip dosyası bir klasöre çıkarılmıştır. İsmi 2017-09-07-raspbian-stretch.img olan bir dosya elimizde var olacaktır. (Dosya ismi indirilen zamana göre değişiklik gösterebilir.) SD kart bilgisayara takılıp Win32 Disk Imager veya benzeri bir program ile açılır. Klasör simgesinden .img dosyası, bu simgenin yanından da SD kartın takılı olduğu sürücü ismi seçilir. Ardından write butonu ile dosyalar SD karta yazdırmaya başlanır. (Şekil 5.2)



Şekil 5.2 İşletim Sisteminin SD Karta Yüklenmesi

İşletim sistemi SD karta yazıldıktan sonra kart Raspberry Pi'ye takılıp Raspberry Pi çalıştırılır. Kullanılan Raspberry Pi 3'de ise internet bağlantısı kablosuz olarak yapılabilir.

Ardından bilgisayardan SSH bağlantısı ile bağlantı sağlayarak işletim sistemi ayarları yapılmaya başlanır. İlk kullanıcı adı pi şifre ise raspberry olacaktır.

```
sudo raspi-config
```

Yukarıdaki komut ile işletim sistemi ayarlarının olduğu ekran açılır. Bu ekranda öncelikle Advanced Options menüsünden Expand Filesystem seçilerek Raspberry Pi'nin tüm SD kartı kullanması sağlanır.Tekrar Advanced Options menüsünden Memory Split seçilir. Burada GPU bölümüne 16 değeri verilir. Ana menüden Finish seçilerek Raspberry Pi kurulumu tamamlanır.

5.3.2 MPICH kurulumu

Öncelikle bilgisayardan Raspberry Pi'ye SSH bağlantısı sağlanır. Ardından aşağıdaki komut ile paketler güncellenir.

```
sudo apt-get update
sudo apt-get upgrade
```

Bazı programlar gerektirdiği için Fortran aşağıdaki komut ile yüklenir.

```
sudo apt-get install gfortran
```

MPICH yüklenecek klasör oluşturulur ve MPICH kaynak kodu bu klasöre indirilir. Daha sonra dosya klasöre çıkarılır.

```
mkdir /home/pi/mpich2
cd ~/mpich2
wget http://www.mcs.anl.gov/research/projects/mpich2/downloads/
tarballs/1.4.1p1/mpich2-1.4.1p1.tar.gz
tar xfz mpich2-1.4.1p1.tar.gz
```

Aşağıdaki komutlar ile derlenmiş dosyaların oluşturulacağı klasörler oluşturulur ve yapılandırma işlemine başlanır. Daha sonra yükleme tamamlanır.

```
sudo mkdir /home/rpimpi/
sudo mkdir /home/rpimpi/mpich2-install
mkdir /home/pi/mpich_build
cd /home/pi/mpich_build
sudo /home/pi/mpich2/mpich2-1.4.1p1/configure
-prefix =/home/rpimpi/mpich2-install
sudo make
sudo make install
```

Aşağıdaki komut ile oluşturulan dosyalar yola eklenir.

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

Aşağıdaki komutlar ile MPICH programının yüklenip yüklenmediği kontrol edilir ve uygulama çalıştırılarak program test edilir.

```
which mpicc
which mpiexec
cd ~
mkdir mpi_testing
cd mpi_testing
```

Raspberry Pi'nin IP adresi ifconfig komutu ile alınarak machinefile adlı bir dosyaya aşağıdaki komut ile yazılır. Daha sonra program çalıştırılır.

```
nano machinefile
mpiexec -f machinefile -n 2 ~/mpich_build/examples/cpi
```

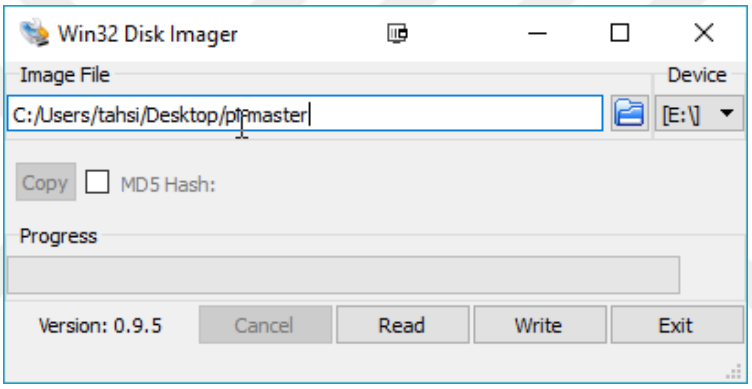
Başarılı bir kurulum aşağıdaki gibi bir çıktı verecektir. (Şekil 5.3)

```
pi@raspberrypi:~/mpi_testing $ mpiexec -f machinefile -n 2 ~/mpich_build/examples/cpi
Process 0 of 2 is on raspberrypi
Process 1 of 2 is on raspberrypi
pi is approximately 3.1415926544231318, Error is 0.0000000008333387
wall clock time = 0.000676
```

Şekil 5.3 MPICH Kurulum Testi

5.3.3 Diğer nodlara kurulum

Öncelikle kurulum yaptığımız ana Raspbbery Pi'den SD kart çıkarılıp bilgisayara takılır. Ardından Win32 Disk Imager programı açılır. Klasör simgesinden .img dosyasının kaydedileceği yer ve isim seçilir. Klasör simgesinin sağından SD kartın takılı olduğu sürücü seçilir. Read butonu ile dosyaya yazdırma işlemi başlatılır. (Şekil 5.4)



Şekil 5.4 SD Karttan Okuma

Ardından işletim sistemi kurulum adımlarında anlatıldığı gibi yeni SD kartı bilgisayara takarak bu .img dosyası yeni SD karta yazdırılır. Yazdırma işlemi tamamlandıktan sonra yeni SD kart Raspberry Pi ile çalıştırılarak test edilir.

5.3.4 Nodlar arası bağlantı

Nodlar arasında bağlantı sağlamak için SSH bağlantısı kurulmuştur. Bu bağlantıyı kurmak için ilk kurulum yapılan Raspberry Pi çalıştırılır ve bilgisayardan Raspberry Pi'ye SSH bağlantısı gerçekleştirilir. Bağlantı sağlanacak diğer Raspberry Pi'de çalıştırılarak ikisinin de IP adresleri not edilir.

Aşağıdaki komutlarla SSH anahtarları oluşturulur. Bu adımda gelen uyarılarda istenirse şifre yazılabilir.

```
1 cd ~
2 ssh-keygen -t rsa -C "raspberrypi@raspberrypi"
```

Oluşturulan anahtarlar diğer makineye kopyalanır. Bu komutta 192.168.0.28 olan yere diğer makinenin IP adresi yazılır. Gelen uyarılardan ilkine "yes" ikinci uyarıya ise "raspberry" yazarak işlem tamamlanır.

```
1 cat ~/.ssh/id_rsa.pub | ssh pi@192.168.0.28
2 "mkdir .ssh;cat >> .ssh/authorized_keys"
```

Aşağıdaki komutla diğer makineye bağlanıp bağlantı test edilir.

```
1 ssh pi@192.168.0.28
```

Bir önceki bölüm ve bu bölümdeki işlemler tekrarlanarak kümeye yeni makineler eklenebilir.

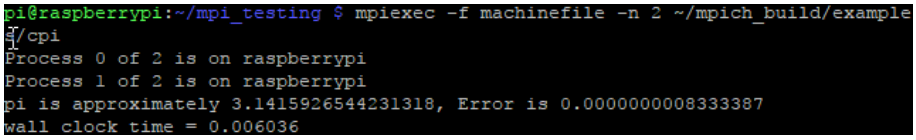
MPICH yüklemesi tamamlanırken kullanılan test fonksiyonu ile makineler test edilir. İlk kurulum yapılan Raspberry Pi üzerinde bir machinefile oluşturulur ve Raspberry Pi'lerin ip adresleri bu dosyaya eklenir.

```
1 cd /home/pi/mpi_testing
2 nano machinefile
```

Aşağıdaki komutla örnek program çalıştırılarak küme test edilir.

```
mpiexec -f machinefile -n 2 ~/mpich_build/examples/cpi
```

Başarılı bir kurulum aşağıdaki gibi bir çıktı verecektir. (Şekil 5.5)



Şekil 5.5 Kümede Uygulama Çalıştırma

Uygulama 5.1

Belirtilen sütun ve satır sayılarına sahip iki matrisin çarpımını yapalım. Bunun için öncelikle rastgele değerlerle bir matris oluşturulur. Daha sonra işlemci sayısına göre gönderilecek ortalama satır sayısı hesaplanır. Bu satırlar işlemcilere MPI_Send fonksiyonu ile gönderilir. Gönderilen değerler MPI_Recv fonksiyonu ile alınarak çarpım işlemi uygulanır ve tekrar ilk işlemciye gönderilir. İlk işlemci gelen değerleri ekrana yazdırır ve program sonlandırılır.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

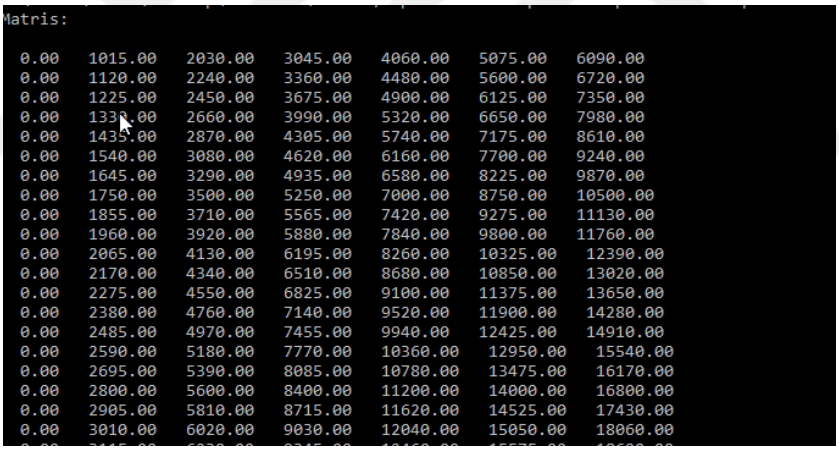
//Satır Sayısı
#define NRA 62
//A matrisi sütun sayısı
#define NCA 15
//B matrisi sütun sayısı
#define NCB 7

int main (int argc, char *argv[])
{
//Değişkenler oluşturuluyor
int numtasks,taskid,numworkers,source,dest,mtype;
int rows,averow, extra, offset,i, j, k, rc;
double a[NRA][NCA],b[NCA][NCB],c[NRA][NCB];
MPI_Status status;
//MPI başlatılıyor ve işlemci küme ve sırası alınıyor
```

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
numworkers = numtasks-1;
//Ana işlemcinin görevi
if (taskid == 0){
for (i =0; i<NRA; i++)
for (j =0; j<NCA; j++)
a[i][j] = i+j;
for (i =0; i<NCA; i++)
for (j =0; j<NCB; j++)
b[i][j] = i*j;
//Matris diğer işlemcilere gönderiliyor
averow = NRA/numworkers;
extra = NRA%numworkers;
offset = 0;
mtype = 1;
for (dest =1; dest<=numworkers; dest++)
{
rows = (dest <= extra) ? averow+1 : averow;
MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
MPI_COMM_WORLD);
MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
offset = offset + rows;
}
//Hesaplanan değerler diğer işlemcilerden alınıyor
mtype = 2;
for (i =1; i<=numworkers; i++)
{
source = i;
MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
MPI_COMM_WORLD, &status);
}
//Değerler yazdırılıyor
printf("Matris:\n");
for (i =0; i<NRA; i++)
{
printf("\n");
for (j =0; j<NCB; j++)
printf("%6.2f ", c[i][j]);
}
}
//Diğer işlemcilerin görevleri
if (taskid > 0)
{
//Değerler alınıyor
mtype = 1;
MPI_Recv(&offset, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&a, rows*NCA, MPI_DOUBLE, 0, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, 0, mtype, MPI_COMM_WORLD, &status);
//Çarpım işlemi yapılıyor
```

```
76 for (k =0; k<NCB; k++)
77 for (i =0; i<rows; i++)
78 {
79 c[i][k] = 0.0;
80 for (j =0; j<NCA; j++)
81 c[i][k] = c[i][k] + a[i][j] * b[j][k];
82 }
83 //Değerler ana işlemciye gönderiliyor
84 mtype = 2;
85 MPI_Send(&offset, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
86 MPI_Send(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
87 MPI_Send(&c, rows*NCB, MPI_DOUBLE, 0, mtype, MPI_COMM_WORLD);
88 }
89 //MPI işlemlerini bitiriyoruz
90 MPI_Finalize();
91 return 0;
92 }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 5.6)



Şekil 5.6 Uygulama 5.1 Sonucu

Uygulama 5.2

f(x) = x^2 fonksiyonunun Trapezoidal kuralı yardımıyla integralini hesaplayalım. Bunun için Trapezoidal kuralını işleyen bir fonksiyon oluşturulur. Uygulamada verilen aralık işlemci sayısına göre eşit parçalara bölünür. Bu parçalar Trapezoidal fonksiyonunda işlenerek ilk işlemciye gönderilir. İlk işlemcide gelen değerleri toplayarak integralin sonucuna ulaşılır.

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

float f(float x);
float Trapezoidal(float aa, float bb, int nn, float h);

//Kullanılacak f fonksiyonu
float f(float x) {
return x*x;
}

//Trapezoidal Kuralı
float Trapezoidal(float aa, float bb, int nn, float h){
float integral;
float x;
int i;
integral = (f(aa) + f(bb))/2.0;
x = aa;
for (i = 1; i <= nn-1; i++)
{
x + = h;
integral + = f(x);
}
integral * = h;
return integral;
}

int main(int argc, char** argv)
{
//Gerekli değişkenlerimizi oluşturuyoruz
int prosesrank,p,nn,source;
float h,aa,bb,integral,toplam;
int dest = 0;
int tag = 50;
MPI_Status status;
//Başlangıç Noktası 3
float a = 3.0;
//Bitiş Noktasu 5
float b = 5.0;
//n 6 alıyoruz
int n = 6;
//MPI başlatılıyor ve işlemci küme ve sırası alınıyor
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &prosesrank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
//Aralığı n eşit parçaya bölüyoruz ve bunları paylaştırıyoruz
h = (b-a)/n;
nn = n/p;
aa = a + prosesrank*nn*h;
bb = aa + nn*h;
//Aralıkların alanlarını hesaplıyoruz
integral = Trapezoidal(aa, bb, nn, h);
// Ana işlemcimizin bize sonuçları vermesini sağlıyoruz
if (prosesrank == 0) {
toplam = integral;
```

```
56 //İşlemcilerden sonuçları alıyoruz
57 for (source = 1; source < p; source++) {
58 MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,MPI_COMM_WORLD, &status);
59 toplam + = integral;
60 }
61 //Sonucu yazdırıyoruz
62 printf("%f\n",toplam);
63 }else{
64 //Değerleri gönderiyoruz
65 MPI_Send(&integral, 1, MPI_FLOAT, dest,tag, MPI_COMM_WORLD);
66 }
67 //MPI işlemlerini bitiriyoruz
68 MPI_Finalize();
69 return 0;
70 }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 5.7)

```
C:\Users\tahsi\Desktop\Örnekler\AlanHesaplama>mpiexec -np 4 AlanHesaplama
18.148149
```

Şekil 5.7 Uygulama 5.2 Sonucu

Uygulama 5.3

Monte Carlo metodu; bir birim kare içine bir çeyrek daire çizilirse birim kare içinde rasgele alınan bir noktanın çeyrek daire içine düşme olasılığı pi / 4'tür. Bu noktalardan daire içinde olanların sayısı toplam nokta sayısına bölünerek pi / 4 değeri dolayısıyla pi değeri hesaplanır. Bu metoda göre pi sayısını hesaplayalım.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <mpi.h>
5
6 #define N 1E8
7 #define d 1E-8
8
9 int main (int argc, char* argv[])
10 {
11 //Gerekli değişkenlerimizi oluşturuyoruz
12 int rank, size, error, i, result =0, sum =0;
13 double pi =0.0, begin =0.0, end =0.0, x, y;
14 //MPI başlatılıyor ve işlemci küme ve sırası alınıyor
```

```
15 MPI_Init (&argc, &argv);
16 MPI_Comm_rank (MPI_COMM_WORLD, &rank);
17 MPI_Comm_size (MPI_COMM_WORLD, &size);
18 //Tüm işlemler senkronize ediliyor
19 MPI_Barrier(MPI_COMM_WORLD);
20 srand((int)time(0));
21 //Her işlemci hesaplamanın kendine ait kısmını yapacak
22 for (i =rank; i<N; i+ =size)
23 {
24 x =rand()/(RAND_MAX+1.0);
25 y =rand()/(RAND_MAX+1.0);
26 if(x*x+y*y<1.0)
27 result++;
28 }
29 //Gelen değerler toplanıyor
30 MPI_Reduce(&result, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
31 //Tüm işlemler senkronize ediliyor
32 MPI_Barrier(MPI_COMM_WORLD);
33 //Ana işlemci çıkan sonucu ekrana yazdırıyor
34 if (rank ==0){
35 pi =4*d*sum;
36 printf("Pi =%0.7f\n", pi);
37 }
38 //MPI işlemlerini bitiriyoruz
39 MPI_Finalize();
40 return 0;
41 }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 5.8)

```
Pi=3.1417511
```

Şekil 5.8 Uygulama 5.3 Sonucu

Uygulama 5.4

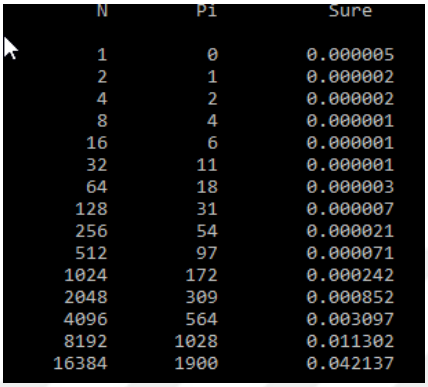
1’den 20000’e kadar olan asal sayıların toplamını bulan programı yazalım. Bunun için gerekli değerler MPI_Bcast ile işlemcilere gönderildilten sonra gönderilen değerlerin asallığı kendinden küçük değerlere bölünerek kontrol edilir. Son olarak toplam değerler ekrana yazdırılır.

```
1 # include <math.h>
2 include <mpi.h>
3 include <stdio.h>
```

```
 4 include <stdlib.h>
 5 include <time.h>
 6
 7 t main ( int argc, char *argv[] );
 8 t asal_sayilar ( int n, int id, int p );
 9 t main ( int argc, char *argv[] ){
10 //Değişkenler oluşturuluyor
11 int i,id,n,n_factor = 2,n_yuksek = 20000,n_alcak = 1,p,asal,asal_parca;
12 double wtime;
13
14 //MPI yükleniyor
15 MPI_Init ( &argc, &argv );
16 MPI_Comm_size ( MPI_COMM_WORLD, &p );
17 MPI_Comm_rank ( MPI_COMM_WORLD, &id );
18
19 //Ana işlemciden üst başlık yazdırılıyor
20 if ( id == 0 ){
21   printf ( "         N        Pi          Sure\n" );
22   printf ( "\n" );
23 }
24
25 n = n_alcak;
26 while ( n <= n_yuksek ){
27 ^^I//Ana işlemci süreyi tutuyor
28   if ( id == 0 ){
29     wtime = MPI_Wtime ( );
30   }
31   //Veriler işlemcilere gönderiliyor
32   MPI_Bcast ( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );
33   //Parçalar işlemcilerden alınarak toplanıyor
34   asal_parca = asal_sayilar ( n, id, p );
35   MPI_Reduce ( &asal_parca, &asal, 1, MPI_INT, MPI_SUM, 0,MPI_COMM_WORLD );
36   //Ana işlemci gelen verileri ekrana yazdırıyor
37   if ( id == 0 ){
38     wtime = MPI_Wtime ( ) - wtime;
39     printf ( "  %8d  %8d  %14f\n", n, asal, wtime );
40   }
41   n = n*n_factor;
42 }
43 I//MPI sonlandırılıyor
44 IMPI_Finalize ( );
45 return 0;
46
47
48 Asal sayıları hesaplayan fonksiyon
49 t asal_sayilar ( int n, int id, int p ){
50 int i,j,asal,toplam =0;
51 for ( i = 2 + id; i <= n; i = i + p ){
52   asal = 1;
53   for ( j = 2; j < i; j++ ){
54     if ( ( i % j ) == 0 ){
55       asal = 0;
56       break;
57     }
58   }
59   toplam = toplam + asal;
```

```
60 }
61 return toplam;
62
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 5.9)



N	Pi	Sure
1	0	0.000005
2	1	0.000002
4	2	0.000002
8	4	0.000001
16	6	0.000001
32	11	0.000001
64	18	0.000003
128	31	0.000007
256	54	0.000021
512	97	0.000071
1024	172	0.000242
2048	309	0.000852
4096	564	0.003097
8192	1028	0.011302
16384	1900	0.042137

Şekil 5.9 Uygulama 5.4 Sonucu

Uygulama 5.5

RSA şifreleme algoritması 1977 yılında Ron Rivest, Adi Shamir ve Leonard Adleman tarafından oluşturulmuştur. Algoritma genel olarak şu şekilde çalışır:

- P ve Q gibi iki asal sayı seçilir.
- N = P.Q ve φ(N) =(P-1)(Q-1) hesaplanır.
- 1'den büyük φ(N)'den küçük φ(N) ile aralarında asal bir E tamsayısı seçilir.
- Seçilen E tamsayısının mod φ(N)'de tersi alınır, sonuç D gibi bir tamsayıdır.
- E ve N tamsayıları genel anahtarı, D ve N tamsayıları ise özel anahtarı oluşturur.
- Gönderilmek istenen bilgi genel anahtar ile şifrelenir.

Algoritma çok büyük asal sayılar oluşturma ve bu sayıların işleminin zorluğu üzerine oluşturulmuştur. Bu yüzden eğer bu asal sayıları işleyebilecek bir paralel program yazılabilirse teorik olarak bu algoritma ile şifrelenen bilginin şifresi kırılabilir. Bu bilgilerden hareketle verilen bir sayının en büyük asal çarpanlarını hesaplayan bir program

yazalım. Bunun için GMP kütüphanesinin fonksiyonlarından yararlanılarak iki asal sayı bulunmuştur.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <math.h>
5  #include <assert.h>
6  #include <gmp.h>
7  #include <mpi.h>
8
9 Iint main(int argc, char** argv)
10   {
11     int sira, p, i, varis,ikinciasal = 0,bcastStatus,esitlik;
12     mpz_t gecerliasal;
13     unsigned long int carpim;
14     sscanf(argv[1], "%lu", &carpim);
15
16    //GMP kütüphabe değişkenleri
17     mpz_t siradakiasal;
18     mpz_t testFactor;
19     mpz_init(siradakiasal);
20     mpz_init_set_str (siradakiasal, argv[1], 10);
21     mpz_init(testCarpim);
22     mpz_init_set_ui(gecerliasal, 2);
23     mpz_nextprime(siradakiasal, siradakiasal);
24     mpz_t carpimTest;
25     mpz_init(carpimTest);
26
27     //MPI yükleniyor
28     MPI_Request sonuc;
29     MPI_Init(&argc, &argv);
30     MPI_Comm_size(MPI_COMM_WORLD, &p);
31     MPI_Comm_rank(MPI_COMM_WORLD, &sira);
32     MPI_Status status;
33
34     //Eğer bir işlemci çarpımı bulduysa cevap alınıyor
35     MPI_Irecv(&ikinciasal, 1, MPI_UNSIGNED_LONG, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &sonuc);
36     for (i =0 ; i < sira ; i++) {
37       mpz_nextprime(gecerliasal, gecerliasal);
38     }
39
40     while (!ikinciasal) {
41       //Başka bir işlemcinin çarpımı bulup bulmadığı kontrol ediliyor
42       MPI_Test (&sonuc, &bcastStatus, &status);
43       if(bcastStatus) {
44         //Bulunduysa tamamlandı
45         MPI_Wait(&sonuc, &status);
46         break;
47       }
48       for (i =0 ; i < p ; i++) {
49         mpz_nextprime(gecerliasal, gecerliasal);
50       }
```

```
51
52      //Sıradaki asal sayının çarpıma uygun olup olmadığı test ediliyor.
53      for (mpz_set_ui(testCarpim , 2) ; mpz_get_ui(testCarpim) <= mpz_get_ui(gecerliasal); mpz_nextprime(test
54        //Başka bir işlemcinin çarpımı bulup bulmadığı kontrol ediliyor
55        MPI_Test (&sonuc, &bcastStatus, &status);
56        if(bcastStatus) {
57          MPI_Wait(&sonuc, &status);
58          break;
59        }
60        mpz_mul_ui(carpimTest, gecerliasal, mpz_get_ui(testCarpim));
61        esitlik = mpz_cmp_ui(carpimTest, carpim);
62        if (esitlik == 0){
63          //Değer bulundu ve ekrana yazdırılıyor
64          ikinciasal = mpz_get_ui(testCarpim);
65          printf("Birinci %lu ikinci %d \n",  mpz_get_ui(gecerliasal), ikinciasal);
66          fflush(stdout);
67          for (varis = 0 ; varis < p ; varis++) {
68            if (varis != sira) {
69              MPI_Send(&ikinciasal, 1, MPI_UNSIGNED_LONG, varis, 0, MPI_COMM_WORLD);
70            }
71          }
72        }
73      }
74    }
75
76    MPI_Barrier(MPI_COMM_WORLD);
77    MPI_Finalize();
78    return(0);
79  }
```

Program çalıştırıldıktan sonra aşağıdaki şekilde çıktı elde edilir.(Şekil 5.10)

```
Birinci 156979 ikinci 157007
```

Şekil 5.10 Uygulama 5.5 Sonucu

6. SONUÇ VE ÖNERİLER

Bu tez çalışmasında paralel programlamanın ne olduğu, MPI ile paralel programlar geliştirmek için gerekli kütüphanenin kullanımı, bu kütüphanenin fonksiyonları ve clusterlar incelenmiştir. Buna ek olarak MPI ile uygulamalar yazılarak bunlar bilgisayar ortamında test edilmiştir.

Paralel programlamanın birçok avantajı bulunmaktadır. Fakat düzgün kurulmamış bir algoritma ile bu avantajlar dezavantaja dönüşebilir. Bu çalışmadaki uygulamalarda paralel programlar seri programlara göre performans artışı göstermiştir. Fakat hatalı yazılan bir algoritma ile paralel programlar seri programlardan daha yavaş çalışabilir.

Seri olarak yazılan programlar paralel programlara dönüştürülebilir. Bu programlar clusterlar üzerinde test edilebilir. Hatta akademik çalışmalar ULAKBİM'deki cluster üzerinde test edilebilir. Meteoroloji, matematik, fizik ve sair alanlarda kullanılan seri algoritmalar paralel algoritma haline dönüştürülerek daha verimli neticeler elde edilebilir.

KAYNAKLAR DİZİNİ

Akçay, M., Erdem, H. A., 2010, Paralel Hesaplama ve MATLAB Uygulamaları, *Türkiye'de İnternet Konferansı*. URL: http://ab.org.tr/ab10/bildiri/207.pdf.

Amdahl, G. M., 1967, Validity of the single processor approach to achieving large scale computing capabilities, URL: http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf.

Anonim, 1990, The Cilk Project, URL: http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf.

Ataş, Ü., Kandıra, C., 2005, Geleceğin Teknolojisi : Bilgisayar Kümeleri, *Türkiye'de İnternet Konferansı*. URL: http://inet-tr.org.tr/inetconf10/bildiri/41.pdf.

Backus, J., 1978, Can Programming Be Liberated From The Von Neumann Style?, 21, 613–641. DOI: 10.1145/359576.359579.

Batcher, K., 1980, Design of a Massively Parallel Processor, *IEEE Transactions on Computers* 29, 836–840. DOI: 10.1109/TC.1980.1675684.

Blelloch, G. E., Maggs, B. M., 1996, Parallel Algorithms, *Algorithms and theory of computation handbook 2*.

Cox, S., 2012, Steps to make Raspberry Pi Supercomputer, URL: https://www.southampton.ac.uk/~sjc/raspberrypi/pi_supercomputer_southampton.htm.

Flynn, M. J., 1972, Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers* C-21, 948–960. DOI: 10.1109/TC.1972.5009071.

Franke, H., Wu, C.-F., Riviere, M, Pattnaik, P., Snir, M., 1995, MPI programming environment for SP1/SP2, *International Conference on Distributed Computing Systems*, 127–135.

Gelernter, D., 1985, The S/Net's Linda Kernel, URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.6086&rep=rep1&type=pdf.

MPI Forum, 2012, MPI: A Message-Passing Interface Standard, URL: http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

MPI Forum, 1994, MPI: A Message-Passing Interface Standard, URL: http://www.netlib.org/utk/people/JackDongarra/PAPERS/059_1994_mpi-a-message-passing-interface-standard.pdf.

Message Passing Interface Forum, 2009, MPI: A Message-Passing Interface Standard, URL: http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

Ngxande, M., Moorosi, N., 2014, Development of Beowulf Cluster to Perform Large Datasets Simulations in Educational Institutions, *International Journal of Computer Applications* 99, 29–35. DOI: 10.5120/17450-8341.

Schneck, P. B., 1987, Dedication, *The Journal of Supercomputing* 1, 5–6.

Slotnick, D., 1982, The Conception and Development of Parallel Processors-A Personal Memoir, *Annals of the History of Computing* 4, 20–30. DOI: 10.1109/MAHC.1982.10003.

Von Neumann, J., 1945, First Draft of a Report on the EDVAC, URL: http://www.virtualtravelog.net/wp/wp-content/media/2003-08-TheFirstDraft.pdf.

Yeo, C. S., Buyya, R., Pourreza, H., Eskicioglu, R., Graham, P., Sommers, F., 2006, Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers, 521–551. DOI: 10.1007/0-387-27705-6_16.

Zhang, W., 2008, Linux Virtual Server for Scalable Network Services, URL: http://www.linuxvirtualserver.org/ols/lvs.pdf.