KOCAELİ ÜNİVERSİTESİ * FEN BİLİMLERİ ENSTİTÜSÜ

ALGILAYICI AĞLARI İÇİN YÖNLENDİRME ALGORİTMASININ PARALEL ANCOR İLE GERÇEKLENMESİ

YÜKSEK LİSANS

Bilgisayar Mühendisi İlker Mustafa MANAP

Anabilim Dalı : Bilgisayar Mühendisliği

Danışman: Yrd. Doç. Dr. H. Engin DEMİRAY

KOCAELİ, 2008

KOCAELİ ÜNİVERSİTESİ * FEN BİLİMLERİ ENSTİTÜSÜ

ALGILAYICI AĞLARI İÇİN YÖNLENDİRME ALGORİTMASININ PARALEL ANCOR İLE GERÇEKLENMESİ

YÜKSEK LİSANS TEZİ

Bilgisayar Mühendisi İlker Mustafa MANAP

Tezin Enstitüye Verildiği Tarih : 6 Kasım 2008

Tezin Savunulduğu Tarih : 26 Kasım 2008

Tez Danışmanı	Üye	Üye
Yrd. Doç. Dr. H. Engin DEMİRAY	Prof. Dr. Ali OKATAN	Doç. Dr. Yaşar BECERİKLİ

KOCAELİ, 2008

ÖNSÖZ

Mobil uygulamaların yaygınlaşması ve kullanıcı sayısındaki artış, kablosuz ağlarda[1] yeni yaklaşımların getirilmesini zorunlu kılmıştır. Çalışmamda, yeni ortaya atılan bir yönlendirme algoritmasının[2, 3], daha etkin bir şekilde simülasyonunun yapılabilmesi için zorunlu olan paralelleştirilmesi işlemini gerçekleştirdim. Daha büyük parametrelerle gerçekleştirilebilecek simülasyonların, araştırmacılara algoritmalarını iyileştirmede yardımcı olacağını düşünüyorum.

Tezimin hazırlanması sırasında bana manevi destek olan eşim Serap Manap'a, algoritma analizi ve tasarlanması sırasındaki yardımlarından dolayı Deniz Demiray ve kardeşim Cevat Manap'a, son olarak çalışmam sırasında yaptığı yönlendirmeler ve tavsiyeleri için danışmanım Yrd. Doç. Dr. H. Engin Demiray'a teşekkürü bir borç bilirim.

İÇİNDEKİLER

ŞEKİLLER DİZİNİ

SİMGELER

Kısaltma	**Açıklaması**
ANCOR	: Ant Colony Routing
ccNuma	: Cache Coherent Non-Uniform Memory Access
CSMA/CA	: Carrier Sense Multiple Access with Collision Avoidance
GPS	: Global Positioning System
MIMD	: Multiple Instruction Multiple Data
MISD	: Multiple Instruction Single Data
MPI	: Message Passing Interface
NUMA	: Non-Uniform Memory Access
OPDM	: Optimal Pheromone Distribution Model
PANCOR	: Parallel ANCOR
PDA	: Personal Digital Assistant
PVM	: Parallel Virtual Machines
RF	: Radio Frequency
SIMD	: Single Instruction Multiple Data
SISD	: Single Instruction Single Data
SoC	: System on Chip
SPMD	: Single Instruction Multiple Data
TLB	: Translation Lookaside Buffer

ALGILAYICI AĞLARI İÇİN YÖNLENDİRME ALGORİTMASININ PARALEL ANCOR İLE GERÇEKLENMESİ

İlker MUSTAFA MANAP

Anahtar Kelimeler: Paralel Hesaplama Sistemleri, Algılayıcı Ağlar, Yönlendirme Simülasyonu

Özet: Bu çalışmada, algılayıcı ağlarında yönlendirme algoritmalarından ANCOR[2, 3] için paralel bir model önermekteyiz. Algılayıcı ağları[1], genelde çok sayıda düğüme sahip ağlardır. Algoritmanın seri uygulaması[2, 3], algılayıcı ağlarının gerektirdiği simülasyon boyutlarına çıkamamaktadır. Önerdiğimiz modelle, düğüm sayısı bir sürecin kapsayabileceği kadar bir hafıza alanı ile, karınca sayısı ise simülasyona eklenebilecek sistem sayısı ile kısıtlanmaktadır. Algoritmanın daha büyük parametrelerle simülasyona sokulabilmesi, karınca kolonisi kullanılarak gerçekleştirilen seri modelin doğal hayata daha uygun bir yapıda modellenebilmesini sağlamıştır.

PARALLEL IMPLEMENTATION OF SENSOR NETWORK ROUTING ALGORITHM ANCOR

İlker MUSTAFA MANAP

Keywords: Parallel Computing Systems, Sensor Networks, Routing Simulation

Abstract: We are proposing a parallel model for the sensor network routing algorithm ANCOR[2, 3]. Sensor networks[1] generally have large number of nodes. The serial implementation[2, 3] of the algorithm cannot be applied to the simulations where large number of nodes and data packets. Our model limits the number of nodes with the memory available to the master process and limits the number of data packets with the number of slave computers. Running the simulation with larger parameters helps to better simulate the natural behavior of ants movements, resulting in more efficient routing of data packets.

1 GİRİŞ

Karınca kolonisi optimizasyonu[4], gerçek karıncaların davranışlarından esinlenen bir optimizasyon yöntemidir. Bir karınca kolonisinde yönetici merkez kavramı bulunmadığından, kararlar pozitif geri beslemelerle ve yerel değişikliklerin toplam etkisi ile alınmaktadır. Karınca kolonisi ile optimizasyon yaklaşık 20 yıldır araştırma konusudur. Özellik seçme[5], veri madenciliği[6] gibi çeşitli uygulamalarının yanında, karınca kolonileri ile algılayıcı ağları üzerinde yönlendirme[2, 3, 5, 7] üzerine az sayıda çalışma bulunmaktadır. Yeni yayınlanmış olan ANCOR[2, 3] gibi çalışmalarda ortaya konulan seri algoritmalar, seri algoritmaların sınırlarından dolayı, doğal yaşamın tam olarak modellenmesinde yetersiz kalmaktadırlar. Bu çalışma ile, karınca kolonileri kullanılarak algılayıcı ağlarında yönlendirme algoritmalarından olan ANCOR[2, 3]'un paralel modellemesi gerçeklenmiştir.

2 ALGILAYICI AĞLARI

Taşınabilir elektronik cihazlar son yıllarda oldukça popüler olmuş ve insan hayatına dizüstü bilgisayar, PDA, GPS cihazları, cep telefonları, müzik çalarlar şeklinde girmişlerdir. Günümüzde, ticari malzemeler kullanılarak, bir cüzdan boyutunda elektronik cihazların yapılması mümkün olmaktadır. Bu küçük cihazlarda, Microsoft Windows ya da Linux işletim sistemlerinin özel sürümleri çalışabilmektedir. Kablosuz bağlantı özellikleri bulunan bu cihazlara, bulunduğu ortamı algılayabilecek ısı, nem gibi algılayıcılar eklenebilmektedir. Böylece, geniş alana yayılmış pek çok algılayıcı, birbirlerine kablosuz olarak erişebilmekte ve kablosuz algılayıcı ağlarının[1, 8] ilk örnekleri de ortaya konmuş olmaktadır.

Bir algılayıcı düğümü; algılayıcı, hesaplayıcı, iletişim, tetikleyici ve enerji kaynağından oluşur. Bu parçalar, genellikle bir ya da birkaç elektronik kart üzerine yerleştirilmiş olup, en fazla birkaç santimetreküplük bir hacim içine sığabilirler. Düşük güç tüketimine sahip elektronik devreler ve ağ teknolojileri ile oluşturulan bir algılayıcı düğümü, 2 AA boyutunda pil kullanılarak %1 düşük çalışma oranı ile yaklaşık üç sene çalışabilmektedir.

Bir kablosuz algılayıcı ağı, birbirleri ile kablosuz kanallardan haberleşerek bilgi paylaşan ve beraber çalışan çok sayıda düğümden oluşur. Kablosuz algılayıcı ağlar, doğal hayatı izleme, askeri gözetleme, arama kurtarma, yapı sağlığı gözetlemesi, akıllı evler ve hatta hasta gözetlemesinde kullanılmaktadır.

Belirli bir amaç için oluşturulan kablosuz algılayıcı ağın ilk kurulumundan sonra, düğümler uygun ağ yapısını kendi kendilerine oluştururlar. Daha sonra, düğümlerde bulunan algılayıcılar, akustik, sismik, manyetik ortam bilgilerini toplamaya başlarlar. Bilgi toplama işlemi, ya belirli zaman aralıklarında, ya da belirli bir eylemin gerçekleşmesi durumunda tetiklenebilir. Düğümlerin konumları, küresel konumlama sistemi kullanılarak elde edilebileceği gibi, yerel konum hesaplama algoritmaları da

kullanılabilir. Ağdan elde edilecek bilgilerle, izlenen nesne ya da ortama ait genel görünüm oluşturulur. Kablosuz algılayıcı ağların temel felsefesi, kısıtlı yeteneklere sahip düğümlerin birlikte çalışarak karmaşık görevleri gerçekleştirebilecek olmasıdır.

Bir kablosuz ağ kullanım senaryosunda, kullanıcılar, kablosuz ağa sorgularını, baz istasyonlarından gönderebilirler. Baz istasyonları, kullanıcı ve kablosuz algılayıcı ağ arasında arayüz sağlar. Bu haliyle kablosuz algılayıcı ağ, dağıtık bir veritabanı gibi çalışır.

2.1 Algılayıcı Ağların Yapıtaşları

2.1.1 Donanım

Düğümleri oluşturan donanımın temel parçaları, farklı teknolojilerdeki ilerlemelerle sürekli gelişmekte ve değişmektedir.

Öncelikle, Entegre Üzerinde Sistem (System-on-Chip) teknolojisi ile, bir bilgisayar sistemi tamamen bir entegre üzerinde gerçeklenmiştir. Atmel, Intel ve Texas Instruments gibi şirketlerin sağladığı SoC tipi gömülü işlemciler, UC Berkeley'deki motes, UCLA'daki Medusa ve WINS düğümlerinde kullanılmaktadır.

İkinci olarak, ticari radyo frekans devreleri kısa mesafe kablosuz haberleşmeyi çok düşük enerji harcaması ile sağlayabilmektedir. RF Monolithics, Chipcon, Conexant Systems ve National Semiconductor firmalarının ürünleri motes, Medusa ve WINS düğümlerinde kullanılmaktadır. Yüzlerce kilobit hızlara çıkabilen bu ticari radyo ürünleri, paket aktarımı ve alımı işlemleri için 20 mW'tan az enerji harcamaktadırlar.

Üçüncü olarak, mikro-elektro-mekanik sistemler teknolojisi ile, aynı CMOS entegresi içine çok sayıda algılayıcının yerleştirilebilmesi sağlanmıştır. Termal, akustik, sismik, manyetik, elektromanyetik algılayıcılar, optik, kimyasal ve biyolojik alıcılar, hız ölçüm cihazları, güneş radyasyonu algılayıcıları, basınç ölçerler gibi algılayıcılar ticari olarak alınabilmektedir. Yukarıdaki algılayıcılar kullanılarak akustik mesafe ölçme, hareket takibi, titreşim algılama ve doğal hayatı izleme gibi çok farklı alanlarda uygulamalar geliştirilmesi mümkündür.

Yukarıdaki teknolojiler gelişmiş paketleme yöntemleri kullanılarak küçük bir algılayıcı düğüm içine algılama, hesaplama, haberleşme ve enerji birimlerinin yerleştirilmesini sağlamıştır.

2.1.2 Kablosuz haberleşme

Kablosuz algılayıcı ağları, donanım teknolojilerinin yanında kablosuz iletişim teknolojilerindeki gelişmelerden de etkilenmektedir. Kablosuz yerel alan ağları için ilk standart, 802.11 protokolü 1997 yılında yayınlanmıştır. Bu standart daha sonra daha yüksek veri hızı ve CSMA/CA mekanizması için ortam erişim kontrolü eklenerek 802.11b olarak yükseltilmiştir. Dizüstü bilgisayarlar ve PDA'lar için tasarlanmış olsa da, 802.11 protokolü ilk kablosuz algılayıcı ağ denemelerinde kullanılmıştır. 802.11 protokolünün yüksek enerji gereksinimi ve çok yüksek veri aktarım hızları, kablosuz algılayıcı ağları için uygun değildir. Bu nedenle, enerji tüketimi ve veri hızı açısından kablosuz algılayıcı ağlarına daha uygun bir ortam erişim kontrolü protokolü geliştirilmesi için çeşitli araştırma grupları çalışmalar yapmışlardır. 802.15.4 protokolü ile kablosuz algılayıcı ağların ihtiyaçlarına uygun düşük güç tüketimi, düşük veri hızına sahip bir kablosuz kişisel alan ağı standardı yayınlanmıştır.

Kablosuz ağlarda yönlendirme teknikleri, kablosuz algılayıcı ağlarda önemli bir araştırma konusu olmuştur. İlk kablosuz algılayıcı ağlarda genellikle kablosuz hareketli ağlar için kullanılan yönlendirme protokolleri kullanılmıştır. Bu protokoller, yüksek güç gereksinimleri nedeniyle kablosuz algılayıcı ağlar için uygun değildir. Bu protokollerin değiştirilerek kablosuz algılayıcı ağlara uygun hale getirilmesi ve yeni yönlendirme teknikleri yoğun olarak araştırılan konulardır.

2.1.3 Ortak işaret işleme

Düğüm tarafından alınan ham ortam verisi işlendikten sonra ortaya kullanılabilir veri çıkmaktadır. Ham verinin düğüm üzerinde işlenmesi ve sadece işlenmiş verinin kullanıcıya aktarılması gerekmektedir. Hesaplama için gereken enerji, kablosuz haberleşme için gereken enerjiden daha azdır. Bu nedenle, çok miktarda veriyi merkeze taşıyıp işlemek yerine, düğümlerin veriyi işleyip, sadece sonucu merkeze göndermesi çok daha az enerji harcanmasını sağlayacaktır.

Bilgi birleştirme, ortak işaret işleme için önemli konulardan birisidir. Gözetlenen ortamın etkileri nedeniyle, düğümlerden okunan verilerde hatalar olabilir. Bilgi birleştirme yöntemleri ile, birden fazla düğümden alınan bilgiler kullanılarak ortam gürültüsünün veriden ayrıştırılması sağlanmaktadır.

2.2 Kablosuz Algılayıcı Ağları Uygulamaları

Kablosuz algılayıcı ağları uygulamaları veri toplama uygulamaları ve hesap ağırlıklı uygulamalar olarak iki temel grupta toplanabilir.

2.2.1 Veri toplama uygulamaları

Doğal Ortam Çalışmaları : Doğal ortam çalışmaları, kablosuz algılayıcı ağlarının en yaygın kullanım bulduğu alanlardan birisidir. Uygulamalarda, gözlenen varlıkların genellikle biyofiziksel ve biyokimyasal özellikleri algılanır. Çoğu senaryoda, minimum, maksimum ve ortalama bulma gibi veri birleştirme işlemlerinin ye aldığı basit işaret işleme yeterli olmaktadır.

Çevre Gözetleme : Bu uygulamalarda, çok geniş alanlara yerleştirilen çok sayıda düşük maliyetli algılayıcılar kullanılır. Orman yangını alamı, sel baskını alarmı, toprak nem miktarı izleme, güneş radyasyonu haritalama gibi işlerde kullanılmıştır.

2.2.2 Hesap ağırlıklı uygulamalar

Binalarda Yapısal Bütünlük İzleme : Kamu yapılarının yapısal bütünlüğünü izlemek uzun zamandır hem endüstride hem de akademik alanda araştırma konusu olmuştur. Yapıların bütünlük kontrolü görsel inceleme, akustik yayım, ultrasonik testler ve radar tomografi gibi geleneksel yöntemlerle yapılmaktadır. Kablosuz algılayıcı ağların ortaya çıkmasıyla, binalarda yapısal bütünlük izleme için daha yeni, binalara zarar vermeyen ve ucuz yöntemler kullanmak mümkün olmuştur. Yapısal bütünlük izleme uygulamalarında, algılayıcıların topladığı ham veri miktarı çok fazladır. Bu nedenle, algılayıcıların sadece gerekli olan bilgileri aktarması, sistemin çalışma süresinin uzatılması açısından önemlidir.

Ağır Sanayi Uygulamaları : Montaj hatları gibi endüstriyel uygulamalarda algılayıcılar fazlasıyla kullanılmaktadırlar. Kablosuz algılayıcıların kullanılmasıyla sorun çıktığında bakım yapılması mümkün olmaktadır. Endüstriyel uygulamalarda, zorlu şartlarda bile güvenilir çalışma önemlidir. Makinalardan kaynaklanacak olan radyasyon, mikroişlemci arızalarına ya da kablosuz iletişimde problemlere yol açabilir. Ayrıca ortam ısı ve nem değerlerinin çok değişken olması, güvenilir donanımların kullanılmasını gerektirir. Endüstriyel uygulamalar genellikle yüksek hesaplama gücü gerektiren karmaşık işaret işleme yöntemleri kullanılır.

3 PARALEL SİSTEMLER

Bilgisayar bilimlerinde sürekli olarak elde bulunandan daha fazla hesaplama gücüne ihtiyaç duyulmaktadır. Bu hesaplama gücü, sayısal modelleme ve mühendislik problemlerinin simülasyonunda yoğun olarak kullanılmaktadır. Bu hesaplama ve simülasyonlar, çok büyük veri kümeleri üzerinde tekrarlı işlemlerin gerçekleştirilerek sonuçlar üretmesini gerektirirler. Hesaplamaların uygun bir zaman aralığında tamamlanması beklenir. Üretim yapılan bir sistem düşünüldüğünde, uygun süre saniyeler, en fazla dakikalar olabilir. Hesaplaması 2 hafta süren bir simülasyon, tasarımcının zamanını fazlasıyla harcayabilir. Makul zamanda biten simülasyonlar, tasarımcılara hata yaptıklarında telafi etme ya da modeli en iyileştirme açısından fayda sağlarlar. Sistemler karmaşıklaştıkça, simülasyonları da daha uzun zaman alacaktır. Bazı problemler belirli bir zaman diliminde çözülmek zorundadır. Ertesi gün için yapılacak hava tahmininin hesaplaması 2 gün sürdüğünde, sonuç anlamsız olacaktır.

Bilgisayarla hava tahmini[9], güçlü bilgisayarların yaptığı işe verilen örneklerden biridir. Atmosfer, üç boyutlu hücreler şeklinde modellenir. Her bir hücre için ısı, basınç, nem, rüzgar hızı, rüzgar yönü belirli zaman aralıkları için, bir önceki zaman dilimindeki değerler kullanılarak hesaplanır. Her bir hücre için, zamanda ileri doğru bu hesaplar tekrar tekrar yapılır. Burada simülasyonu anlamlı kılan özellik, hücre sayısıdır. Örneğin tüm atmosferi kapsayan, hücre boyu $1mil \times 1mil \times 1mil$ olan ve 10 mil yüksekliğe kadar alanı içeren bir simülasyonu düşünelim. Verilen büyüklüklerle, yaklaşık 5×10^8 hücre bulunduğu görülür. her bir hesaplamanın 200 kayar nokta işlemi gerektirdiğini varsayalım. Bir zaman diliminde 10^{11} kayar nokta işlemi gerekecektir. Hava tahminini 10 gün için 10'ar dakikalık dilimlerle yapmaya kalktığımızda, 10^4 zaman dilimi ve toplamda 10^{15} kayar nokta işlemi gerekecektir. 1 Gflop'luk (10^9 kayar nokta işlemi / saniye) bir bilgisayar ile simülasyon 10^6 saniye, yaklaşık 11 günde tamamlanabilecektir. Simülasyonun 10 dakika içinde tamamlanmasını istersek, 1.7 Tflop (1 Tflop, 10^{12} kayar nokta işlemi / saniye) gücünde bir sisteme ihtiyaç vardır.

Hesaplama hızını artırmanın bir yolu, bir problem üzerinde birden fazla işlemci kullanılmasıdır. Büyük problem parçalara bölünür ve her bir parça farklı bir işlemciye verilir. Bu tarz uygulamalar geliştirmeye paralel programlama adı verilir. Hesaplama platformu (paralel bilgisayar), özel tasarlanmış çok işlemcili bir bilgisayar olabileceği gibi, bağımsız bilgisayarların biraraya getirilmesiyle de oluşturulabilir. Bu yaklaşım, performansta büyük artış sağlar. Buradaki temel fikir, n tane bilgisayarın, bir bilgisayarın n katı kadar hesaplama gücü sağlayabileceğidir. Böylece problem teorik olarak $\frac{1}{n}$ zamanda çözüme kavuşturulabilir. Problemler genellikle küçük parçalara kolaylıkla bölünemezler. Veri transferi ve hesaplamaların senkronizasyonu gibi işlemler nedeniyle, hesaplamanın yapılamadığı zaman dilimleri bulunmaktadır. Bu zaman dilimlerinin varlığı, bir problemin n bilgisayarda $\frac{1}{n}$ zamanda bitirilememesine neden olur. Buna rağmen problemin yapısına ve paralelleştirilebilmesine göre problemin çözüm süresi önemli ölçüde kısalacaktır.

3.1 Paralel Bilgisayar Tipleri

Paralel programlama, üzerinde çalışacağı uygun bir platforma ihtiyaç duyar. Bu platform, çok sayıda işlemciye sahip bir bilgisayar ya da çok sayıda birbirine bağlı bilgisayardan oluşabilir.

3.1.1 Paylaşımlı hafızaya sahip çokişlemcili sistemler

Geleneksel bir bilgisayar, hafızadaki bir programı çalıştıran bir işlemciden oluşur. Her bir hafıza gözüne, adres adı verilen sayısal değer ile konumlanılır.

Tek işlemcili modeli genişletmenin doğal yollarından biri, çok sayıda işlemcinin çok sayıda hafıza modüllerine bağlı olduğu modeldir. Bu modelde her bir işlemci, herhangi bi hafıza bölgesine erişebilir. İşlemcilerle hafıza modülleri arasındaki bağlantı bir ağ (interconnection network) ile sağlanır. Paylaşımlı hafızalı çok işlemcili sistem bir tek adres uzayı kullanır. Bütün hafıza içinde her bir alanın ayrı bir adresi vardır ve bu adrese sistemdeki her bir işlemci tarafından erişilebilir.

Çoğu tek işlemcili sistem, sanal hafıza yöntemini kullanır. Sanal hafıza, gerçek sistemde bulunan hiyerarşik ve farklı hızlara sahip hafızaların yüksek hızlı ana hafıza gibi görünmesini sağlayan yöntemdir. Hafıza bölgelerini otomatik olarak daha yavaş

hafıza olan diske aktarıp, gerektiğinde diskten ana hafızaya yükleyebilirler. Burada her bit hafıza alanı için iki adres bulunur: İşlemci tarafından üretilen sanal adres, gerçek adres alanına ulaşmak için kullanılan gerçek adres. Sanal adres ile gerçek adres arasında dönüşüm yapabilmek için donanımsal bir tablo (Translation Look-aside Buffer, TLB) kullanılır. Sanal hafıza modeli, paylaşımlı hafızaya sahip çok işlemcili bilgisayarlara uygulanabilir. Böylece, her hafıza alanının gerçek bir adresi varken, her bir işlemci farklı sanal adreslerle bu hafıza bölgelerine erişebilir.

Paylaşımlı hafızaya sahip çokişlemcili bilgisayarlar programlar çalıştırılırken, her bir işlemcinin çalıştıracağı uygulama hafızada saklanır. Her uygulamanın erişeceği veri de hafızada saklanır. Programcının çalışan uygulamayı ve uygulamanın kullanacağı veriye erişim şeklini tanımlaması birkaç farklı şekilde olabilir. Özel paralel programlama yapıları ve paylaşımlı değişkenlere ve paralel kod bloklarının tanımlanabilmesine olanak sağlayan yeni bir paralel progamlama dili geliştirilebilir. Derleyici, programcının tanımlamasından, çalışabilir uygulamayı oluşturacaktır. Bir diğer yöntem ise, iş parçacıkları (thread) kullanarak yüksek seviyeli dillerle tanımlanan uygulama bloklarının her bir işlemci tarafından çalıştırılabilmesini sağlamaktır. Bu uygulama blokları, paylaşımlı ortak hafızaya erişim yeteneğine sahiptir.

Programcı açısından, paylaşımlı hafızaya sahip çokişlemcili sistemler, paylaşımlı hafızaya erişimde sağladıkları kolaylıklar nedeniyle daha kolay programlanabilen sistemlerdir. Fakat böyle bir sistemde bütün işlemcilerin paylaşımlı hafızaya hızlı erişimini sağlamak, donanımsal olarak sağlanması zor bir özelliktir. Bu tarz büyük sistemlerde, işlemcilerin kendilerine fiziksel olarak yakın hafıza bölgelerine daha hızlı erişimi sağlayan hiyerarşik ya da dağıtık hafıza yapıları bulunur. Bu tarz hafıza erişim yapısına sahip sistemlere değişken hızlı hafıza erişimli sistemler (non-uniform memory access systems, NUMA) adı verilir.

Geleneksel tek işlemcili sistemlerde hızlı önbellek, yakın zamanda erişilen hafıza bölgelerinin kopyalarını tutar. Bu durum, çok işlemcili sistemlerde hafıza bölgelerine erişimde farklı hızların otaya çıkmasına neden olur. Ayrıca, farklı önbelleklerde bulunan aynı veri bölgesi içeriğinin birbirinin aynı olmasının sağlanması da ciddi bir

problemdir. Bir işlemcinin önbelleğinde değiştirilen veri bölgesinin, diğer işlemcilerin önbelleklerinde yer alıyorlarsa değiştirilmesi gerekir. Bu durumu donanımsal olarak çözen sistemler (SGI-Altix) bulunmaktadır. Bu sistemlere önbellek tutarlı değişken hızlı hafıza erişimli sistemler (cache coherent NUMA) denir.

3.1.2 Mesaj geçişli çoklu bilgisayar

Paylaşımlı hafızaya sahip çokişlemcili sistemler, özel olarak tasarlanmış bilgisayar sistemleridir. Çok işlemcili sistemler için bir alternatif ise, birbirinden farklı bilgisayar sistemlerinin bir ağ bağlantısı ile oluşturdukları sistemdir. Sistemi oluşturan her bir bilgisayar, bir işlemci ve diğer sistemler tarafından erişilemeyen yerel hafızaya sahiptir. Bir mesaj geçişli çoklu bilgisayar sisteminde hafıza, sistemi oluşturan bilgisayalara dağıtılmıştı ve her bir bilgisayar kendi hafıza adres alanına sahiptir. Her bir işlemci sadece kendi yerel hafıza alanına erişebilir. Bilgisayarlar arasındaki ağ bağlantısı, işlemcilerin birbirlerine mesaj gönderebilmelerini sağlar. Mesajların içinde, diğer işlemcilerin hesaplamalarını yapabilmesi için gereken veriler bulunur. Bu şekilde çalışan sistemlere mesaj geçişli çoklu bilgisayar (message passing multicomputer) adı verilir.

Mesaj geçişli çoklu bilgisayar sistemlerini programlarken, problemin aynı anda çalışabilecek parçalara bölünmesi gerekmektedir. Programlama yapılırken, özel paralel programlama dilleri ya da genişletilmiş seri programlama dilleri kullanılır. Genellikle kullanılan yöntem ise, geleneksel programlama dillerini bir mesaj geçiş kütüphanesi ile birlikte kullanarak uygulama geliştirmektir. Problem, birbirinden bağımız çalıştırılabilecek alt parçalara bölünür. Bu alt parçalara süreç adı verilir. Süreçler, bağımsız bilgisayarlarda çalıştırılabilirler. Örneğin 10 süreç ve 10 bilgisayar var ise, her bir bilgisayar bir süreç çalıştırır. Sistemi oluşturan bilgisayarlardan daha fazla süreç var ise, her bir bilgisayar birden fazla süreci zaman paylaşımlı olarak çalıştırabilir. Süreçler birbirlerine mesaj göndererek haberleşirler. Mesaj gönderme, verinin ve sonuçların süreçler arasında paylaşılmasının tek yoludur.

Mesaj geçişli çoklu bilgisayar sistemleri, paylaşımlı hafızaya sahip çokişlemcili sistemlere göre daha kolay ölçeklenirler. Ağa eklenecek yeni sistemler, mesaj geçişli çoklu bilgisayar sistemini kolaylıkla daha yüksek performans seviyelerine taşır.

Mesaj geçişli çoklu bilgisayar sisteminin programlanması, paylaşımlı hafızaya sahip çokişlemcili sistemlerin programlanmasına göre daha karmaşıktır. Mesaj geçişini sağlayan kod parçalarının programcı tarafından özellikle yazılması gerekmektedir. Süreçler veriyi paylaşamazlar. Bu nedenle verinin süreçten sürece kopyalanması gerekmektedir. Verinin kopyalanmak zorunda olması, çok büyük veri setlerinde çalışmak zorunda olan problemlerde ciddi performans sorunlarına yolaçmaktadır.

Mesaj geçişli çoklu bilgisayarlarda, veriye aynı anda erişim sırasında kontrol gerektirmeyen özel mekanizmalar vardır. Bu mekanizmalar, paralel uygulamanın çalışma süresini ciddi oranlarda kısaltır. Mesaj geçişli yapının kullanılmasındaki en çarpıcı sebep, bu yapının bir ağa bağlı bilgisayar grubuna direkt olarak uygulanabilmesidir. Pek çok özel tasarlanmış paylaşımlı hafızaya sahip çokişlemcili sistem, sürekli artan işlemci hızı ve işlemci mimarisi nedeniyle oldukça kısa ömürlü olmaktadır. Genellikle hızlı ve yeni bir işlemci kullanmak, işlem hızı ve enerji tüketimleri açısından çok sayıda eski işlemci kullanmaktan daha efektif olacaktır. Yeni işlemciler eski işlemcilerden daha fazla performansı, eski işlemcilere göre daha düşük maliyetlerle sağlayacaktır.

3.1.3 Flynn sınıflandırması

Tek işlemcili bir bilgisayar sisteminde, çalıştırılacak olan komut dizileri bir programdan oluşturulur. Flynn[10], bilgisayarlar için bir sınıflandırma yapmış ve tek işlemcili bilgisayarları tek komut dizisi-tek veri dizisi (single instruction single data, SISD) olarak adlandırmıştır.

Genel amaçlı çokişlemcili bir sistemde, her bir işlemci farklı bir uygulamayı çalıştırabilir. Her bir işlemci, ayrı bir programdan kaynaklanan komut dizilerini çalıştırabilir. Bu tip bilgisayarlar, çoklu komut dizisi-çoklu veri dizisi (multiple instruction multiple data MIMD) olarak adlandırılır.

Aynı komut dizisinin birden fazla işlemci üzerinde birbirinden farklı veri setlerine uygulanabildiği sistemlere tek komut dizisi-çoklu veri dizisi (single instruction multiple data, SIMD) adı verilir.

Flynn sınıflandırmasında dördüncü model, çoklu komut dizisi-tek veri dizisi (multiple instruction single data, MISD) olarak adlandırılır. Bu model, hataya dayanıklı özel sistemler için geçerli olabilir.

3.2 Mesaj Geçişli Çoklu Bilgisayarların Mimari Özellikleri

3.2.1 Statik ağ mesaj geçişli çoklu bilgisayarları

Mesaj geçişli çoklu bilgisayarlar, mesajların taşınması için bir bağlantıya ihtiyaç duyarlar. En çok kullanılan ağ yapısı, statik ağ yapısıdır. Statik ağ yapısında bilgisayarlar arasında fiziksel bağlar bulunur. Her bilgisayarda, işlemci, hafıza ve diğer bilgisayarlarla bağlantı için bir arayüz kartı bulunur.

Ağ Kriteri: Ağ tasarımında dikkat edilmesi gereken bant genişliği, ağ gecikmesi, bağlantı sayısı ve maliyet gibi birkaç nokta vardır. Bant genişliği, birim zamanda transfer edilebilecek bit sayısını gösterir. Ağ gecikmesi, bir mesajın ağa aktarılması için geçen süredir. Haberleşme gecikmesi, yazılım kaynaklı gecikmeler, arayüz gecikmeleri ve mesajın aktarılması için geçen sürenin toplamıdır. Mesaj gecikmesi ya da başlama süresi, boyu 0 olan bir mesajın aktarılması için geçen süreye denir.

İki bilgisayar arasında gönderilen mesajın gecikmesi hesaplanırken, mesajın geçtiği bağlantı sayısı önem kazanır. Çap, ağ üzerinde birbirine en uzak iki bilgisayar arasında bulunan bağlantı sayısıdır. Çap, en kötü durumda ortaya çıkacak gecikmenin hesaplanması için kullanılır.

Paralel bir problemin efektif çözümünde, ağın yapısının önemi çok büyüktür. Ağın çapı, bir mesajın ağ üzerinde dolaşacağı en uzun mesafeyi gösterir ve paralel algoritmanın iletişim alt sınırını oluşturur. Örneğin sayıların sıralanmasında, her bir sayının ağdaki bir bilgisayarda tutulduğunu varsayalım. Amaç, bilgisayarların numaralarına göre sayıları yer değiştirmektir. Burada en kötü senaryo, bir sayısı, ağ üzerinde en uzakta olan bilgisayara taşımaktır. Eğer ağ çapı d ise, herhangi bir sıralama algoritmasının iletişim alt sınırı en kötü durumda en az d adım gerektiğinden, d sayıda haberleşme adımından az olamaz.

Tamamen Bağlı Ağ: Tamamen bağlı ağlarda, her bilgisayarın sistemdeki diğer bilgisayarlarla arasında bir bağ bulunur. n sayıda bilgisayar, her bilgisayarda bulunan $n-1$ sayıda bağlantı ile diğer bilgisayarlara bağlıdır. Toplamda $\frac{n\times(n-1)}{2}$ bağlantı bulunur. Bu yapı, sadece küçük sayılar için uygulanabilir. n büyüdükçe, her bilgisayar içinde $n-1$ bağlantı sağlamak hem mühendislik hem de maliyet açısından zorlaşacaktır. Bu durumda, kısıtlı bağlantılı statik ağlar (Doğru/halka, ızgara, hiperküp, ağaç) kullanılır.

Doğru/Halka: Bir sıra halinde bulunan bilgisayarlarda bağlantılar sadece komşu bilgisayarla sınırlıdır. Sıra yapısı, iki uçtaki bilgisayarlar arası bağlantı kurularak halka yapısına dönüştürülebilir. Her bilgisayarda, sağ ve solundaki bilgisayarlarla bağlantısını sağlayan iki bağlantı bulunur. Sıra şeklinde olan yapılarda çap $n-1$, halka şeklinde olan yapılarda ise $\lfloor \frac{n}{2} \rfloor$'dir.

Ağın tamamen bağlı yapıları sağlayamadığı durumlarda, direkt bağlı olmayan iki bilgisayar arasında iletişim sağlanabilmesi için bir yönlendirme algoritmasına ihtiyaç duyulur. Sıra ya da halka yapısında yönlendirme algoritması, sağa ya da sola ilerlemek şeklindedir. Farklı ağ yapılarında kullanılan değişik yönlendirme algoritmaları bulunmaktadır.

Izgara: Izgara şeklindeki ağ yapılarında her bilgisayar, iki boyutlu bir matrisin elemanları gibi ya da bir ızgaranın birleşim noktalarına yerleşmiş gibi görünür. Kenarlarda bulunan boş uçlar diğer uç ile birleştirilirse, torus yapısı oluşur.

Izgara ve torus yapıları, kolay kurulabilmeleri ve genişleme kapasiteleri nedeniyle çok popülerdir. Üç boyutlu ızgaralar, her bir bilgisayarın üç boyutta ikişer bağlantı yapmasıyla oluşturulabilir. Üç boyutlu ızgaralar, mühendislik ve bilimsel problemlerin çözümü için daha uygun ağ ortamı sağlarlar.

Ağaç: Ağaç ağlarında ilk bilgisayar kök olarak adlandırılır. Her bilgisayar, kendisinden daha aşağıda bulunan iki bilgisayara, kendisinden daha yukarıda bulunan bir bilgisayara bağlıdır. Ağ kökten aşağıya doğru genişler. Kökün altındaki ilk seviyede iki

bilgisayar bulunur. İkinci seviyede dört bilgisayar, j. seviyede ise 2^j bilgisayar bulunur. Ağın tamamında ise $2^{j+1}-1$ bilgisayar bulunur. Bu tip ağlarda bütün dallar doldurulmuş ise tamamlanmış ikili ağaç olarak adlandırılır. Yükseklik, kökten en alt seviyeye kadar olan bağlantı sayısıdır. Ağaç şeklindeki ağlarda ağın tamamlanmış olması ya da her bilgisayarın altında iki bağlantı olması şart değildir.

Ağaç yapıları, parçala ve ele geçir algoritmaları için uygundur. Ağaç yapılarında köke doğru trafik artmaktadır. Bunu engellemek için şişman ağaç ağları kullanılır. Şişman ağaç ağlarında köke doğru çıkıldıkça bağlantı sayısı artırılır.

Hiperküp: d-boyutlu hiperküp ağında her bilgisayar, diğer bilgisayarlara, her boyutta bir bağlantı olacak şekilde bağlıdır. d-boyutlu hiperküp ağında, her bir bilgisayara d-bitlik bir adres verilir. Adresteki her bir bit, bir boyuta karşılık gelir ve 0 ve 1 değerlerini alabilir. Üç boyutlu hiperküpte her bilgisayara 3 bitlik adres verilir. 000 adresli bilgisayar, 001, 010 ve 100 adresli bilgisayarlara bağlıdır. Burada dikkat edilmesi gereken, bir bilgisayara bağlı olan diğer bilgisayarlarının adreslerinin, bilgisayarın adresinden sadece bir bit farklı olmasıdır. Bu özellik boyut sayısı yükselse de aynı şekilde korunur.

Hiperküplerin önemli avantajlarından birisi, ağ çapının $log_2 n$ kadar olmasıdır. Hiperküplerin özel yapılarından dolayı minimum mesafe, çıkmaza sürüklenmeyen yönlendirme algoritması bulunmaktadır.

3.2.2 Haberleşme yöntemleri

Kaynak düğümden hedef düğüme bir mesaj gönderirken en uygun durum, kaynaktan hedefe direkt bir bağlantının bulunmasıdır. Çoğu sistemde bir mesajın kaynaktan hedefe varabilmesi için yönlendirilmesi gerekmektedir. Mesajlar transfer edilirken iki temel yöntem kullanılır: Paket anahtarlama, devre anahtarlama.

Devre anahtarlamada, kaynaktan hedefe kadar kesintisiz bir yol oluşturulur. Mesaj bu yol üzerinden aktarılır. Mesaj aktarımı tamamlanana kadar yol bozulmadan bekler. Basit telefon şebekesi devre anahtarlama sistemlerine örnek olarak verilebilir. Kurulan

bir telefon bağlantısı, iki uçtan birisi konuşmayı sonlandırana kadar kesilmez. Ağ yapısı büyüdükçe, iki bilgisayar arasındaki bağlantıda aradaki bağlantıların haberleşme için rezerve edilmesi, sistemin bütününün performansını negatif yönde etkiler.

Paket anahtarlamalı sistemde, gönderilecek mesaj paketlere bölünür. Her pakette yönlendirme için gerekli olan kaynak ve hedef adresleri bulunur. Paketlerin taşıyabileceği veri boyları sınırlıdır. Bu nedenle, mesajın boyu paket içindeki veri boyu sınırını geçecek olursa, ağa birden fazla mesaj gönderilmiş olur. Posta sistemi , paket anahtarlamalı sistemler için bir örnektir. Mektuplar, göndericinin posta kutusundan alınarak posta merkezilerine getirilir. Burada posta kutusu uygulama, posta merkezi ise bilgisayarın mesaj tampon bölgesidir. Merkeze alınan mektup, alıcının merkezine gönderilir. Alıcı merkezi de mektubu alıcının posta kutusuna yerleştirir. Bu şekilde gerçekleşen paket anahtarlamalı iletime, depola ve yönlendir (store and forward) adı verilir. Depola ve yönlendir modeliyle çalışan paket anahtarlamalı sistemler eldeki veri paketi yönlendirilirdikten sonra bağlantının farklı paketler için de kullanılabilmesini sağlar. Depola ve yönlendir modeli, hedef ile olan bağlantı sağlansın ya da sağlanmasın, paketlerin bir tamponda bekletilmesini zorunlu tutar. Bu durum, ağ gecikmesinin yükselmesine neden olur. Tampon bölgede saklama gerekliliği, sanal kestirme yönteminin kullanılması ile ortadan kalkar. Sanal kestirme yönteminde, hedef ile bağlantı sağlanmış ise, mesaj tampon bölgeye aktarılmadan direkt olarak hedefe gönderilir. Eğer hedef ile bağantı yoksa, mesajın depolanması için bir tampon bölge kullanılması zorunludur.

Ağ gecikmesini azaltmak ve kullanılan tampon bölgeleri küçültmek amacıyla solucan deliği yönlendirmesi (wormhole routing) Dally and Seitz[11] tarafından ortaya atılmıştır. Depola ve yönlendir metodunda mesaj, hedefle bağlantı sağlandığında bir bütün olarak gönderilir. Solucan deliği yönlendirmesinde mesaj flit adı verilen küçük parçalara ayrılır. Hedef ile kaynak arasındaki bağlantı, flit içindeki her bir bit için bir kablo sağlayabilir. Bu durumda bir flit, paralel olarak aktarılabilir. Bağlantı sağlandığında, gönderilecek olan mesajın sadece başlık kısmı hedefe aktarılır. Mesajın izleyen flitleri aradaki bağlantı sağlandıkça gönderilir. Böylece, flitler, ağ üzerinde dağıtık olarak taşınır. Baştaki flit ilerledikçe, bir sonraki de ilerleyebilmektedir. Düğümler arasında bir istek/onaylama mekanizmasının bulunması flitleri düğümler

arasında hareket ettirmek için gerekmektedir. Bir flit tampon bölgeden ayrılmaya hazır olduğunda, bir sonraki düğüme bir istek gönderir. İstek gönderilen düğüm bir flit tamponu boşaldığında, gönderici düğümden fliti alır. Mesaj aktarımı sırasında mesajın parçalarının birbirine bağlı olması yüzünden aradaki bağlantının korunması gerekmektedir.

3.2.3 Girdi/Çıktı

Bütün bilgisayar sistemlerinin girdi çıktı cihazları ve mekanizmaları bulunmaktadır. Disk sistemleri uygulamaların ve verilerin saklanması için kullanılır. Bir bilgisayarın diskinde bulunan bilgiye diğer bir bilgisayarın erişmek istemesi, ağ üzerinden sözkonusu verinin taşınmasını gerektirecektir. Aynı veriye çok sayıda bilgisayarın erişmek istemesi durumunda ise, performans problemleri başgösterir. Bu durumla başa çıkabilmek için, paralel dosya sistemleri geliştirilmektedir. Paralel dosya sistemlerinde, disk alanı bir grup bilgisayar tarafından tüm sisteme sağlanır. Disk erişimindeki hız, paralel dosya sistemini oluşturan bilgisayar sayısının artırılması ile artar. Lustre[12] ve PVFS[13] en çok kullanılan paralel dosya sistemlerindendir. Bu sistemlerin mimarisi, dosya sistemini sunan bilgisayar grubuna eklenti yapıldıkça tüm paralel hesaplama sisteminin disk erişim hızını artıracak şekilde tasarlanmıştır.

4 MESAJ GEÇİŞLİ PROGRAMLAMA

Bu bölümde mesaj geçişli uygulamaların temel yapısından ve süreçler arasında haberleşmenin nasıl yapıldığından bahsedilecektir. Zamanla eskiyen Paralel Sanal Makinalar[9] (Parallel Virtual Machines, PVM) kütüphanesi, yerini daha geniş kapsamlı olan Mesaj Geçiş Arayüzü (Message Passing Interface, MPI[9]) kütüphanesine bırakmaktadır. MPI, aslında bir kütüphane değil, standarttır. Bu nedenle farklı firmalar standarda uygun MPI gerçeklemeleri çıkarabilirler. Donanım da üreten HP, IBM gibi şirketlerin kendi donanımları için yazılmış MPI standardına uyan kütüphaneleri bulunmaktadır.

4.1 Mesaj Geçişli Programlamanın Temelleri

Paralel uygulamalar, yükek seviyeli bir dile yapılacak kütüphane eklentisi ile yazılabilirler. C ve Python gibi dillere yapılan kütüphane eklentileri ile mesaj geçişli uygulamalar yazmak mümkündür. Kütüphanelerde bulunan komutlar yardımıyla, paralel sistem üzerinde süreçler arası haberleşme yapmak mümkündür. Bu programlama yapısında, hangi süreçlerin çalıştırılacağı, süreçler arasında ne zman mesaj gönderileceği, mesajların içinde hangi bilgilerin olacağını programcının vermesi gerekmektedir. Paralel bir uygulama yazmak için farklı bilgisayarlarda ayrık süreçler başlatabilmek ve süreçler arasında mesaj alıp verebilmek yeterli olmaktadır.

4.1.1 Süreç yaratılması

Paralel programların farklı sayıda işlemci üzerinde testleri sırasında birden fazla süreç bir işlemci üzerinde çalıştırılabilir. Bu durumda işlemci zaman paylaşımı yaparak kendisine verilen süreçleri çalıştırır. Bu şekilde uygulama en hızlı olacak şekilde çalıştırılamaz, ancak programın çok sayıda işlemci üzerinde çalıştırılması test edilmiş olur. Paralel uygulamalarda, bir işlemci üzerinde bir süreç çalıştırılacak şekilde düzenleme yapılarak, sistemin verilen uygulama için en iyi performansı vermesi sağlanır.

Paralel sistemlerde iki türlü süreç yaratılabilir: Statik süreç yaratılması, dinamik süreç yaratılması. Statik süreç yaratılması durumunda, sistemdeki bilgisayarlar uygulama başlatıldığında hangi uygulamayı her bir işlemcide kaç aedt çalıştıracaklarını bilirler. Bu değerler uygulama komut satırından çalıştırılırken verilir. Çoğu paralel uygulamada bir yönetici süreç ile çok sayıda köle süreçler bulunur. Tekli program çoklu veri (Single Instruction Multiple Data, SPMD) modeline göre hazırlanmış uygulamalarda yönetici ve köle süreçleri bir program içine yerleştirilmiştir. Uygulama başlatıldığında, sürecin paralel sistemdeki konumuna (rütbe) göre, uygulamanın yönetici ya da köle kısmı devreye girmektedir. Paralel sistem heterojen mimaride bilgisayarlardan oluşuyor ise uygulamanın her bir mimari için derlenmesi gerekir. Böylece, tüm sistem farklı mimarilerde bilgisayarlardan oluşsa bile, her bilgisayar kendi mimarisine uygun programı çalıştırabilir.

Dinamik süreç yaratmada, uygulamanın çalışması sırasında yeni süreçler uygulama tarafından başlatılır. Süreç yaratma için süreç yapıları ya da sistem çağrıları kullanılır. Süreçler yaratılabildikleri gibi yok edilebilirler. Bu durumda uygulamanın çalışması sırasında sistemde değişen sayılarda süreç aktif olabilir.

4.1.2 Mesaj geçme fonksiyonları

MPI kütüphanesi içinde çok sayıda fonksiyon olmasına rağmen, temel bir paralel uygulamanın yazılabilmesi için az sayıda fonksiyon yeterli olmaktadır. En temel fonksiyonlar mesaj gönderme ve mesaj alma fonksiyonlarıdır. Fonksiyonlara parametre olarak hangi bilgileri gönderecekleri ya da gelen bilgiyi hangi tampon bölgeye aktaracakları verilir.

```
send(parametre listesi)
recv(parametre listesi)
```

Burada send() komutu mesajı gönderecek süreç tarafından çağırılır. Mesaj alacak olan sürecin de recv() komutunu çağırması gerekmektedir. Kullanılacak olan parametreler uygulamanın yazıldığı programlama diline göre farklılık gösterebilir. C dilinde gönderilecek verinin bir tampon bölgede hazırlanıp, gönderim komutu içinde tipinin belirtilmesi gerekir. Python dilinde ise, karmaşık nesnelerden oluşan bir dizi, hiçbir işleme gerek kalmadan gönderim komutuna verilebilir. C kullanarak MPI ile mesaj gönderimde:

```
int x=1;
MPI_Send(&x,1,MPI_INT,1,msgtag,MPI_COMM_WORLD);
```

C kullanarak MPI ile mesaj alımında:

```
int x;
MPI_Recv(&x,1,MPI_INT,0,msgtag,MPI_COMM_WORLD,status);
```

Python kullanarak MPI ile mesaj gönderimde:

```
x=1
mpi.send(x,0)
```

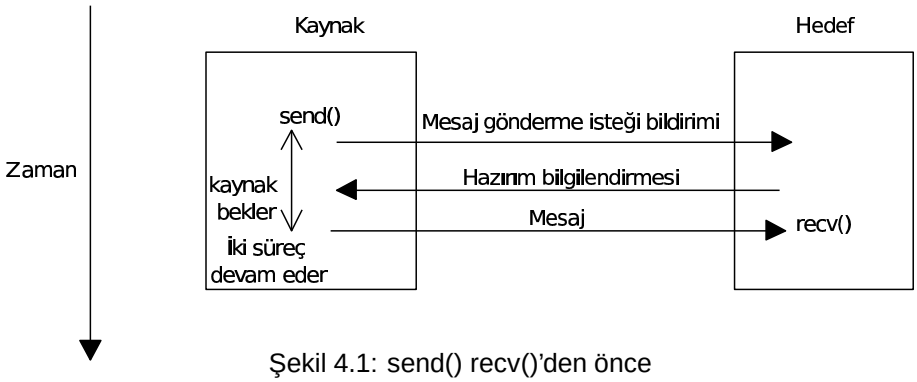
Python kullanarak MPI ile mesaj alımında:

```
cevap,durum = mpi.recv()
```

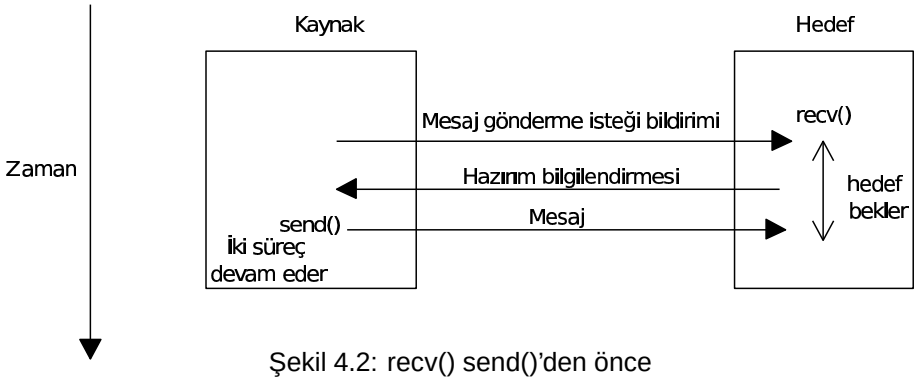
Eşzamanlı Mesaj Gönderme: Eşzamanlı terimi, mesajın gönderimi tamamlanmadan program kontrolünü bırakmayan fonksiyonlar için kullanılır. Burada, mesaj tampon bölgesi kullanılmaz. Eşzamanlı gönderim fonksiyonu, karşı taraftaki uygulama tarafından mesaj alım işlemi başlatılana kadar bekler. Eşzamanlı gönderim ve alım yapan uygulamalarda mesaj değişim işlemi bitene kadar iki süreçte beklemek zorundadır. Bu fonksiyonlar, süreçlerin arasında veri aktarımı ile beraber, paralel uygulamanın akışını da düzenlerler.

Eşzamanlı mesaj alışverişinde üç yollu işaretleşme protokolü kullanılır. Önce kaynak süreç, hedef sürece mesaj gönderme isteğini gönderir. Hedef, mesajı almaya hazır olduğunda hazırım mesajını kaynağa gönderir. Hazırım mesajını alan kaynak, gerçek mesajı hedefe göndermeye başlar.

Şekil 4.1'de kaynak send() kısmına hedeften önce gelir. Bu durumda, kaynak süreci hedef sürecinden cevap alana kadar beklemeye başlar. Eğer kaynak sürecinin mesajlarına ihtiyaç duyan başka süreçler de varsa, bu süreçler de beklemeye geçecektir. Şekil 4.2'de hedef süreci kaynak süreç mesajı göndermeye başlayana kadar bekler. Bu nedenle, send() ve recv() komutlarının süreçler içinde nerede başlayacağı, uygulamanın çıkmaza girmemesi açısından önemlidir. Süreçlerin nasıl bekletileceği, sistemden sisteme farklılık gösterebilir.



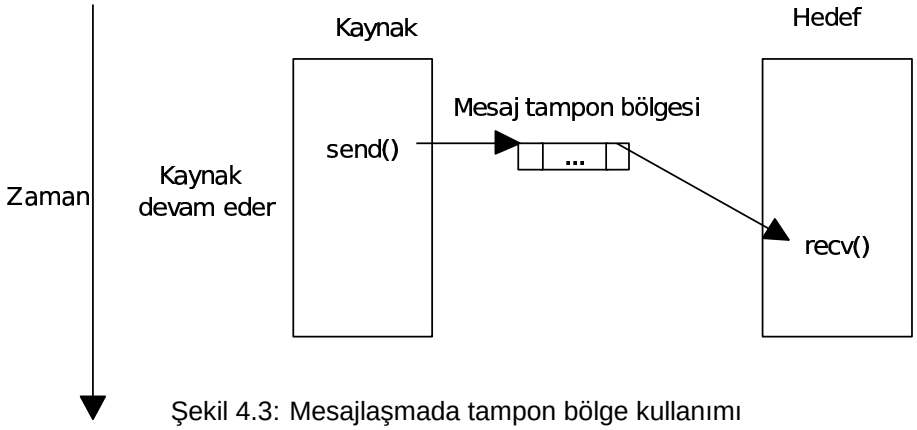
Şekil 4.1: send() recv()'den önce



Şekil 4.2: recv() send()'den önce

4.1.3 Engellenmeyen ve engelleyici mesaj geçişi

Engelleyici terimi, mesaj gönderimi bitmeden fonksiyondan uygulamaya geri dönmeyen anlamında kullanılmaktadır. Eşzamanlı ve engelleyici bu açıdan aynı anlama gelmektedir. Engellenmeyen terimi, mesaj hedef tarafından alınsın ya da alınmasın fonksiyondan çıkarak uygulamaya dönen anlamında kullanılır. Sistemlerde Şekil 4.3'te görüldüğü gibi, gönderilecek mesajı geçici olarak tutan tampon bölgeler bulunur. Tampon bölge, gönderilecek olan mesajın hedef tarafından recv() komutu ile alınana kadar saklanmasını sağlar. Uygulamanın çalışması sırasında, hedef süreç recv() komutuna kaynak sürecin send() komutundan önce gelirse, mesaj tampon bölgesi boştur ve hedef süreci beklemeye geçer. Kaynak tarafında ise, sürecin yerel işlemleri tamamlandıktan ve mesaj tampon bölgeye yerleştirildikten sonra, kaynak süreci hedefin mesajı almasını beklemeden devam edebilir. Böylece uygulamanın toplam çalışma süresi kısalabilir. Pratikte, tampon bölgeler sınırlı büyüklükte olurlar. Uygulamanın çalışması sırasında tampon bölgenin doldurulması mümkündür. Bu durumda hem tampon bölgenin durumunu kontrol etmek hem de hedef sürecin mesajı aldığından emin olabilmek için fazladan mesajlaşma yapılması gerekebilir.



Şekil 4.3: Mesajlaşmada tampon bölge kullanımı

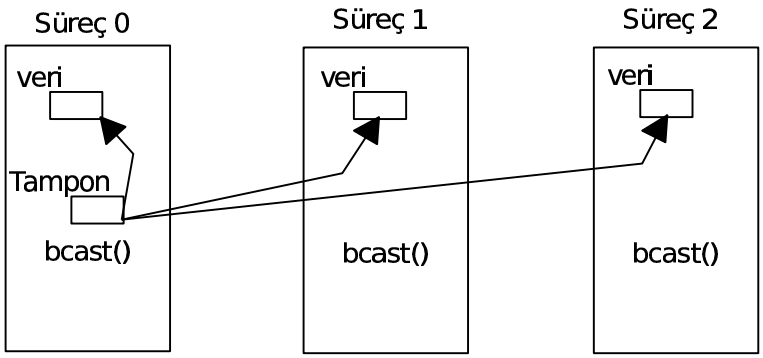
4.1.4 Mesaj seçme

Şimdiye kadar send() komutuna parametre olarak gönderilecek mesaj ve hedef süreç, recv() komutuna da gelen mesajı aktaracağı bellek bölgesi ve mesajın geleceği süreç verildiğini gördük. Hedef süreç, mesajı beklediği süreç dışında gelen mesajların hepsini gözardı edecektir. Bu durumda, gelen tüm mesajların alınabilmesi için süreç yerine özel bir sembol ya da rakam kullanılır. Böylece recv() komutu herhangi bir süreçten gelen mesajı alabilir.

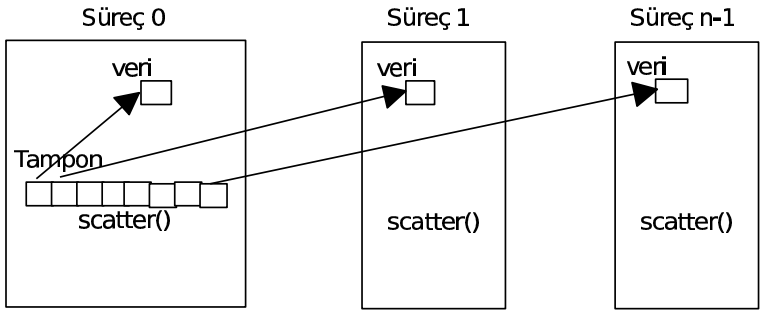
Daha esnek bir kullanım sağlamak amacıyla mesajlar işaretlenebilir. Bu işaret mesajın başlık bilgisine eklenir. İşaret olarak pozitif tamsayı kullanılır. Böylece gönderilen ve alınan mesajların gruplanabilmesiyle beraber, sadece belli bir işarete sahip mesajların alınabilmesi de mümkündür.

4.1.5 Yayma (broadcast), dağıtma (scatter), toplama (gather), indirgeme (reduce)

Yayma, aynı mesajın sistemdeki bütün süreçlere gönderilmesi işlemidir. Yayma işleminde rol alacak olan süreçlerin bir grup oluşturması gereklidir. Yayma işlemi verilen grup içinde gerçekleşir. Şekil 4.4'de süreç 0 yayma işleminin kök sürecidir. Örnekte, kök süreç diğer süreçlere yayılacak olan veriyi tampona aktarır. Şekil 4.4'de SPMD modelindeki gibi bütün süreçler aynı bcast() komutunu çağırırlar. bcast() komutu eşzamanlı, engelleyici bir komuttur. Bütün süreçler bcast() komutunu tamamlayana kadar, yayma işlemine karışmış olan tüm süreçler bekler.



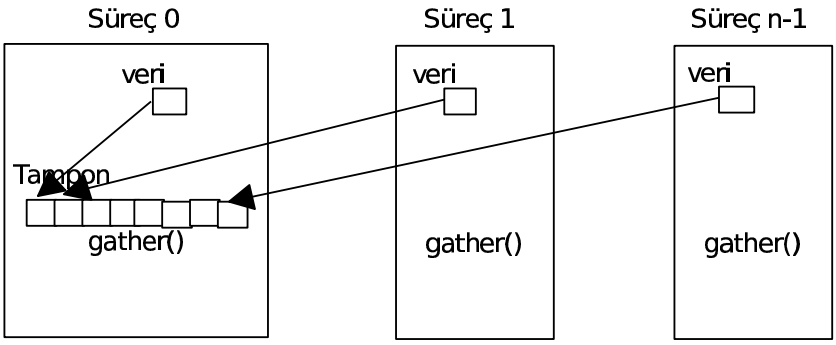
Şekil 4.4: Yayma işlemi



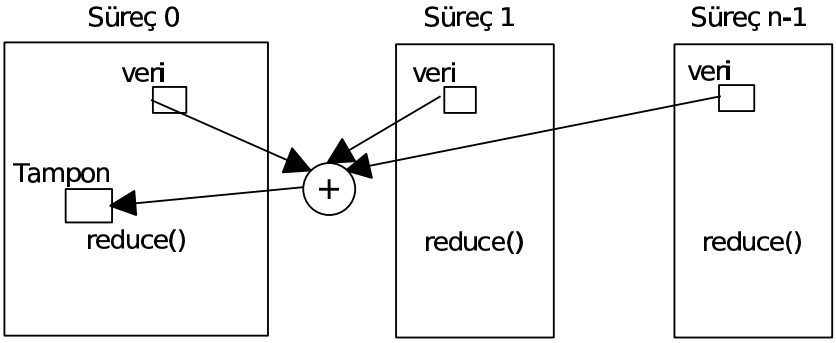
Şekil 4.5: Dağıtma işlemi

Dağıtma terimi, kök süreçte bulunan bir dizinin her bir elemanını ayrı bir sürece gönderme işlemidir. Örneğin dizinin 5. elemanı, 5 numaralı sürece gönderilir. Dağıtma işlemi Şekil 4.5'de gösterilmiştir. Yayma işleminde olduğu gibi, dağıtma işleminde de bir grup süreç işleme dahil olur.

Toplama işlemi, bir sürecin diğer süreçlerden farklı değerleri toplaması şeklinde gerçekleşir. Toplama işlemi, genellikle süreçler tarafından yapılan bir hesaplama işlemi sonrasında kullanılır. Her bir süreç problemin kendisi ile ilgili küçük parçasını hesaplar. Kök süreç, hesaplanan değerleri toplama işlemi ile toplayarak büyük problemin sonucunu çıkarır. Toplama işlemi, yayma işleminin tam tersi olacak şekilde çalışır. Örneğin 5. sürecin hesapladığı değer, kök sürecindeki dizinin 5. gözüne yerleşir. Toplama işleminin nasıl gerçekleştiği, Şekil 4.6'da verilmiştir.



Şekil 4.6: Toplama işlemi



Şekil 4.7: İndirgeme işlemi

Kimi zaman toplama işlemi aritmetik ya da mantıksal operatörlerle birleştirilebilir. Örneğin veriler diğer süreçlerden alınır ve kök süreçte toplama işlemine sokulabilir. Bu tip operasyonlara indirgeme (reduce) adı verilir. Çoğu mesaj geçişli sistemde yayma, dağıtma, toplama ve indirgeme işlemleri için çeşitli yöntemler sağlanmaktadır. İşlemlerin sistemin daha da aşağısında nasıl gerçekleştiği ise, sistemlere bağlı olarak farklılık gösterebilmektedir.

4.2 MPI ile Paralel Programlama

Uzun yıllar boyunca pek çok paralel programlama dili ortaya çıkmış fakat hiçbirisi MPI kadar yaygınlaşma şansı bulamamıştır. MPI kütüphanesi, yüksek seviyeli dillere eklenti olarak gelir ve bu dillere farklı bilgisayarlarda çalışan süreçler arasında mesaj aktarabilme yeteneğini kazandırır. MPI kullanarak programlama genelde C ve FORTRAN kullanarak yapılmaktadır. Burada örneklerimizi C ve Python dillerinde gerçekleyeceğiz.

MPI, C diline direkt kütüphane olarak eklenebilir. Paralel uygulamamızı geliştirirken kullandığımız pyMPI, standart Python yorumlayıcısının kendisinin MPI kütüphaneleri ile beraber derlenmesiyle yaratılan yeni Python yorumlayıcısının içine yerleştirilir.

4.2.1 OpenMPI ve pyMPI paketlerinin kurulumu

OpenMPI[14], birçok araştırma laboratuvarı ve ticari şirketlerin oluşturduğu bir organizasyon tarafından geliştirilen açık kaynak kodlu bir MPI2.0 standardı gerçeklemesidir. Pek çok unix tabanlı sisteme kurulabilir. Kullandığımız Pardus ve CentOS gibi Linux dağıtımları için ayrıca hazırlanmış paketleri bulunmaktadır. Aşağıdaki işlemler, herhangi bir Linux dağıtımında, mpi kütüphanesinin nasıl çalışmaya hazır hale getirileceğini göstermektedir.

Aşağıdaki işlemler bir linux kabuğunda gerçekleştirilir:

```
wget http://www.openmpi.org/software/ompi/v1.2/\
downloads/openmpi-1.2.8.tar.bz2
wget http://surfnet.dl.sourceforge.net/sourceforge/\
pympi/pyMPI-2.5b0.tar.gz
tar jxf openmpi-1.2.8.tar.bz2
cd openmpi-1.2.8
./configure; make ; make install
cd ..
tar zxf pyMPI-2.5b0.tar.gz
cd pyMPI-2.5b0
./configure --with-libs=/usr/local/lib ; make ; make install
```

Yukarıdaki işlemler sonrasında, işletim sistemimizde MPI kullanarak uygulama geliştirmek için gerekli olan bütün kütüphane ve programlama dilleri hazırlanmış olacaktır.

4.2.2 Merhaba dünya uygulaması

Merhaba dünya uygulamasının C sürümü aşağıdadır.

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

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

```
{
    int rutbe;
    int surecsayisi;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rutbe);
    MPI_Comm_size(MPI_COMM_WORLD, &surecsayisi);

    printf("Rütbe %d, Süreç Sayısı %d, Merhaba dünya\n", \
            rutbe,surecsayisi);
    MPI_Finalize();
    return 0;
}
```

Yukarıdaki uygulama merhaba.c adı ile kaydedilir ve aşağıdaki gibi derlenerek ve çalıştırılır:

```
mpicc merhaba.c -o merhaba
mpirun -np 4 merhaba
```

Yukarıda derleme işlemi sırasında, MPI kütüphanesi için gereken eklemeleri mpicc uygulaması kendisi yapmaktadır. Gerekiyorsa başka matematik gibi diğer kütüphaneleri -l parametresi ile derleme işlemine sokabiliriz. Programın çıktısı ise aşağıdaki gibidir:

```
Rütbe 3, Süreç Sayısı 4, Merhaba dünya
Rütbe 0, Süreç Sayısı 4, Merhaba dünya
Rütbe 1, Süreç Sayısı 4, Merhaba dünya
Rütbe 2, Süreç Sayısı 4, Merhaba dünya
```

Aynı uygulamanın Python kodu ise aşağıdaki gibidir:

```
import mpi

rutbe = mpi.rank
surecsayisi = mpi.size

print "Rütbe %d Süreç sayısı %d, Merhaba dünya" % \
  (rutbe, surecsayisi)
```

Python yorumlamalı bir dil olduğundan derleme gerektirmez. Yukarıdaki uygulama ise aşağıdaki gibi çalıştırılır:

```
mpirun -np 4 pyMPI merhaba.py
```

Programın çıktısı ise aşağıdaki gibidir:

```
Rütbe 0 Süreç sayısı 4, Merhaba dünya
Rütbe 3 Süreç sayısı 4, Merhaba dünya
Rütbe 2 Süreç sayısı 4, Merhaba dünya
Rütbe 1 Süreç sayısı 4, Merhaba dünya
```

Yukarıdaki uygulamalarda, 4 süreç yaratılmış ve her bir süreç ekrana istenen metni yazmıştır. Program çıktılarında, rütbelerin hep aynı sırada olmadığı görülür. Paralel sistem her başlatıldığında, süreçlerin komutları gerçekleştirmesi her zaman için aynı sırada olmayacaktır. Bu durumda, sistemin eşzamanlı yürüyebilmesi için engelleyici komutlar kullanılır.

4.2.3 Mesaj gönderme ve alma

Mesaj gönderip alan uygulamanın C sürümü aşağıdaki gibidir:

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

int main(int argc, char** argv)
{
    int rutbe;
    int surecsayisi;
    int i;
    int rakam;
    int koledeki_rakam;

    MPI_Status durum;
    rakam = 1;
    koledeki_rakam = 0;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rutbe);
    MPI_Comm_size(MPI_COMM_WORLD, &surecsayisi);

    if (rutbe == 0)
    {
      for (i = 1; i < surecsayisi; i++)
      MPI_Send(&rakam,1,MPI_INT,i,500,MPI_COMM_WORLD);
    }
    else
    {
      MPI_Recv(&koledeki_rakam,1,MPI_INT,0,500, \
               MPI_COMM_WORLD,&durum);
      koledeki_rakam = rutbe * 10 + koledeki_rakam;
      printf("Rutbe %d, Rakam  %d\n ",rutbe, \
             koledeki_rakam);
    }
```

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

Yukarıdaki uygulama gonder-al.c adı ile kaydedilir ve aşağıdaki gibi derlenerek ve çalıştırılır:

```
mpicc gonder-al.c -o gonder-al
mpirun -np 4 merhaba
```

Programın çıktısı ise aşağıdaki gibidir:

```
Rutbe 3, Rakam  31
Rutbe 1, Rakam  11
Rutbe 2, Rakam  21
```

Aynı uygulamanın Python kodu ise aşağıdaki gibidir.

```
import mpi

rutbe = mpi.rank
surecsayisi = mpi.size
rakam = 1
koledeki_rakam = 0

if rutbe == 0:
  for i in range(1,surecsayisi):
    mpi.send(rakam,i)
else:
  cevap, durum = mpi.recv()
  koledeki_rakam = rutbe *  10 + cevap
  print "Rütbe %d Rakam %d" % (rutbe, koledeki_rakam)
```

Programın çıktısı ise aşağıdaki gibidir:

```
Rütbe 1 Rakam 11
Rütbe 3 Rakam 31
Rütbe 2 Rakam 21
```

5 GENİŞ KARINCA KOLONİLERİNDE DOĞAL YAŞAM

Bütün karınca kolonilerinde hayatın aynı şekilde yürüdüğü genellikle düşünülse de büyük ve küçük karınca kolonilerinin arasında çeşitli farklar bulunmaktadır. Büyük karınca kolonileri, feromon adı verilen bir kimyasal maddenin salgılanması ile haberleşmektedirler.

Mallon[15], karınca sayısındaki azalmanın, bireysel kararlardaki önemi artırdığını göstermiştir. Bu makalede, çok sayıda noktaya sahip algılayıcı ağları uygulama alanı olarak seçildiğinden, bireysel tercihlerin dramatik etkilerinin azaltılması ya da etkisiz kılınması hedeflenmektedir. Bu nedenle, az sayıda bireye sahip karınca kolonileri bu araştırmanın dışındadır. Bu aşamadan sonra karınca kolonisi ile sadece yüksek nüfusa sahip karınca kolonilerinden bahsediyor olacağız.

Büyük bir karınca kolonisi, yeni besin kaynaklarını bulup, besini yuvalarına getiren kestirme yolu bulabilirler ve bu yolu, değişen ortamlara dinamik olarak uydurabilirler. Gerçekte, her bir karınca basit kuralları izleyen bir ajandır. En yüksek feromon değerini takip eder ve yola feromon salgılar. Feromon kimyasal bir salgıdır. Doğada, karınca besin kaynağına ulaşmak için en yüksek feromon değerini takip eder ve yuvasına geri dönerken de feromon salgılayarak düğümlerdeki feromon miktarını yeniler. Bu otokatalitik yöntem, en uygun yolu ortaya çıkarır. Değişik besin kaynaklarına erişim için, bütün karıncalar en yüksek feromon değerini izlemezler. Bazı karıncalar, en uygun yolun dışına çıkarak, yeni besin kaynakları ya da aynı besin kaynağına daha kısa bir yolu ortaya çıkarmaya çalışırlar. Daha kısa bir yol bulunduğunda, daha fazla karınca yeni yolu kullanmaya başlayacağından, yeni yolun feromon değeri eski yolun feromon değerinin üzerine çıkacak, eski yolun feromon değerinin buharlaşma yoluyla azalması ile birlikte, tüm koloni yeni yola uyum sağlamış olacaktır.

6 KARINCA KOLONİSİ İLE YÖNLENDİRME

Literatürde, karınca kolonisi ile algılayıcı ağlarda yönlendirme üzerine az sayıda makale bulunmaktadır. Zhang, makalesi[16] ile, Antnet[17] gibi algoritmaların algılayıcı ağlarında düzgün çalışmadığını söyler. Zhang, var olan metodun ağ üzerinde hedefi bulana kadar çok fazla zaman kaybettiğini, algılayıcı ağlarındaki asimetrik bağlantı yüzünden bu yaklaşımın pratik olmayacağını söyler. Değişik uygulamalar için geliştirilmiş olan karınca kolonisi ile yönlendirme algoritmalarında, karıncalar gezdikleri düğümlerin listesini tutar ve yolun sonunda listede bulunan düğümlerin feromon değerlerini değiştirirler. ANCOR[2] ile önerilen yöntemle, karıncalar, gezdikleri düğümlerin listesini tutmak zorunda kalmadan yönlendirme yapılabilmektedir. Her bir adımda, karıncalar sadece hangi düğümden geldiklerini hatırlar ve üzerinde bulundukları düğümün feromon değerini değiştirirler. Böylece algoritmaya, yönlendirme için daha küçük bir veri paketi yeterli olmaktadır. Burada küçük veri paketinin kullanılmasının amacı, algılayıcı düğümünün veri yayma süresini azaltmaktır. Azalan veri paketi büyüklüğü ve veri yayma süresi, ağın enerji harcamasını azaltarak, toplam çalışma süresini uzatır. Ağ üzerindeki her bir karınca, sadece hangi düğümden geldiğini ve hedefini bilir. Karıncalar, daha önce o düğümden geçen karıncaların bıraktıkları feromon miktarına göre yönlerini belirlerler.

Doğal hayatta karıncaların bıraktıkları feromonun etkisi, mesafe arttıkça azalmaktadır. Ayrıca, bir düğüme gelen karınca, sözkonusu düğümdeki feromon değerini değiştirdiğinde, kendi komşuluğunda bulunan düğümlerin yeni feromon değerinden etkilenmesi gerekmektedir. Demiray[3] makalesinde düğümlerin komşularından etkilenerek feromon değerlerini değiştirebilmelerini modellemiştir. Bu modele göre, yakınındaki düğüme çok yüksek feromon bırakılan düğüm, kendi feromon değerini, yüksek feromon bırakılan düğümdeki feromon miktarına göre yeniden belirlemektedir. Böylece, karıncaların hedefi bulma ve yuvaya dönme sırasında kullandıkları feromon haritası düğümlerin etkilerini de kullanarak oluşturulmaktadır.

6.1 Eşleştirme

Bu bölümde, karınca kolonisi ile algılayıcı ağları arasındaki eşleştirme verilmiştir. Doğal yaşamdaki karınca, algılayıcı ağlarda veri paketi yerine geçer. Aynı şekilde karınca yuvası alıcıyı, besin kaynağı hedefi ve krıncaların kullandığı yol ise verinin izlediği rotayı temsil etmektedir.

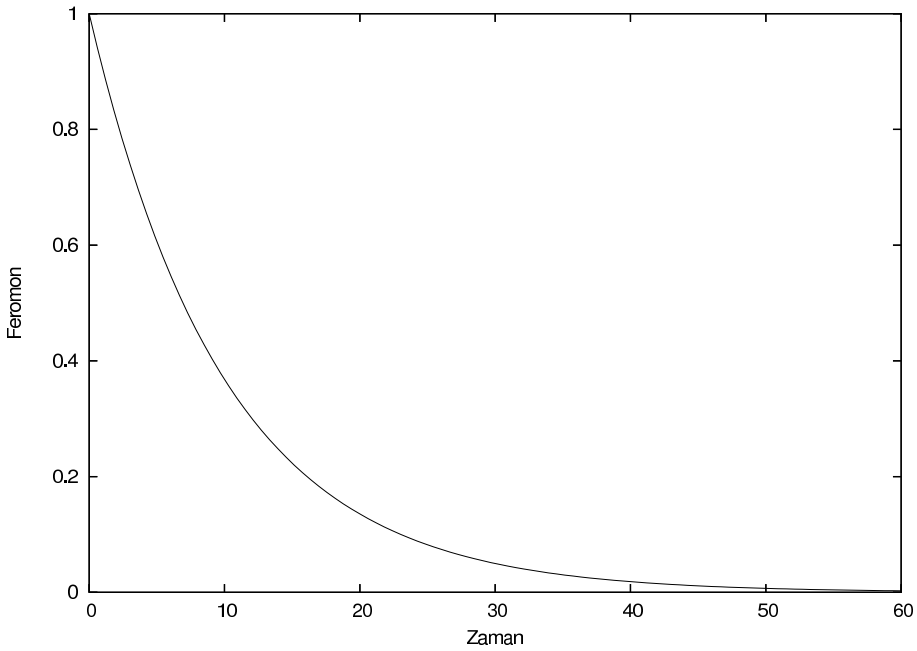
Karıncalar, doğada buldukları besinleri yuvalarına getirirler. Algılayıcı ağlardaki alıcı, doğal yaşamdaki karınca yuvası yerine geçmektedir. Algılayıcı ağlarında varılmak istenen hedef düğüm, doğal yaşamda karıncaların hayatlarını sürdürebilmek için aradıkları besin kaynakları yerine geçmektedir.

Doğal yaşamdaki bir karınca, algılayıcı ağlarında bir sorgu ya da veri paketine karşılık gelmektedir. Her bir algılayıcı düğümü yol üzerindeki durulacak noktalardan birisidir. Bu nedenle her bir düğüm feromon değerini tutarak, bir feromon izi oluşturulmasına yardımcı olur. Karınca kolonisinin yolu, yeri belirlenmiş algılayıcı düğümleri ile gösterilir. Doğal feromon konsepti, yönlendirme algoritmasının temelini oluşturur. Her bir düğüm, erişiminde olan diğer diğer düğümlerin feromon değerlerini içeren bir listeye sahiptir. Düğüme gelen karınca, düğümün yakınında bulunan diğer düğümlerin feromon değerlerini hissederek, bir sonraki adımda nereye gideceğini saptar. Feromon yoğunluğu, düğüme gelen karıncanın izleyeceği yolu seçiminde etkili olan bir parametredir. Gerçek feromonun bir özelliği olan yoğunluk ve buharlaşma, yönlendirme algoitmasında da kullanılmıştır. Algoritma içinde de, düğümlere bırakılan feomon değerleri zaman içinde azalmaktadır.Bu özellik sayesinde, karıncaların sık kullanılmayan komşu düğümlere gitme olasılığı azaltılmaktadır. Feromon yoğunluğunun zamana göre değişimi Şekil 6.1'de gösterilmiştir.

$$y = e^{-\frac{x}{\delta}} \tag{6.1}$$

δ, uygulamaya özel belirlenebilecek ayar parametresidir. Şekil 6.1'de $\delta = 10$ olarak alınmıştır. Aynı matematiksel model, feromon yoğunluğu ve uzaklık arasında da bulunmaktadır.

$$ph_{ij} = e^{-\frac{uzaklık \times zaman}{yogunluk}} \tag{6.2}$$

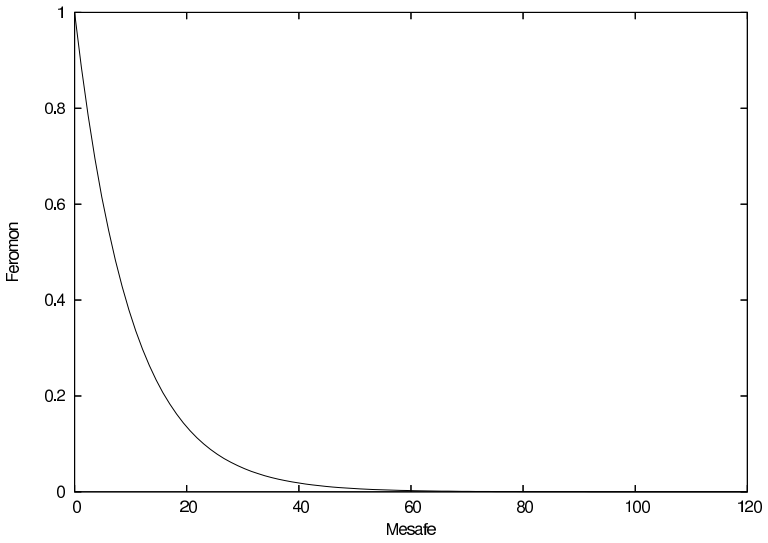


Şekil 6.1: Feromon - Zaman grafiği

Şekil 6.2'de feromon değerinin uzaklığa bağlı olarak azaldığı görülmektedir. Bu durum, gerçek hayatta da aynı şekilde gerçekleşmektedir. i düğümünde algılanan feromon miktarı Denklem 6.1'de verilmiştir. Denklem 6.2'de, ph_{ij}, i düğümünde, j düğümünde bulunan feromon miktarının ne kadarının algılandığını göstermektedir. Uzaklık, iki düğüm arasındaki uzaklıktır. Zaman ise, feromon miktarının zaman içinde buharlaşmasını modeller.Yoğunluk ise, i düğümündeki listede, j düğümünün feromon değeridir.

6.2 Yönlendirme Algoritması ANCOR

Ağ üzerinde bulunan her düğüm, komşu düğümlerin feromon değerlerinin bulunduğu bir tabloya sahiptir. Düğüme gelen karınca, düğümün komşularında bulunan feromon değerlerini kullanarak bi sonraki adımda hangi komşu düğüme geçeceğini saptar. Karınca bir düğüme geldiğinde, düğümün feromon değerini değiştirir. Düğüm, yeni değeri komşularına bildirerek, komşularının tablolarında da yeni feromon değerinin görünmesini sağlar. Her düğüm, kendi feromon tablosunun buharlaştırılma işlemini kendisi gerçekleştirir. Böylece, düğümlerin birbirlerine buharlaşma sonrası yeni feromon değerlerini göndermesine gerek duyulmaz.



Şekil 6.2: Feromon - Uzaklık grafiği

Bütün yönlendirme işlemi üç ana parçadan oluşur:

İlklendirme

Güçlendirme

Yönlendirme

İlklendirme safhasında, yiyecek arayan karıncalar sahaya yayılırlar. Bir karınca yeni bir kaynak bulduğunda, güçlendirme safhası başlar. Kaynağı bulan ilk karınca, yuvaya dönüş yolunu güçlendirir. Güçlendirme işlemi, düğüme maksimum feromon verilerek gerçekleştirilir. Güçlendirilmiş yol oluşturulunca yönlendirme işlemi başlar. Bütün karıncalar feromon değeri açısından en güçlü olan düğümleri kullanarak kaynaklara ulaşmaya çalışırlar. Karıncaların bir kısmı da, güçlendirilmiş yolun dışına çıkarak yeni kaynakların bulunmasına yardımcı olurlar.Karıncaların sahada bulunan diğer kaynakları araştırmasını sağlamak için başka kontrol mekanizmaları da bulunmaktadır. Yönlendirme algoritmasında feromonun çekici ve itici özelliğinin tanımlanabilmesi sağlanmıştır. Etkisiz feromon ”1” ile belirtilir. Çekici feromon $]1..2]$ aralığında tanımlanır. Doğal yaşamda bulunmayan itici feromon ise $[0..1[$ aralığında tanımlıdır. Benzer bir yapı Montgomery [18] tarafından da kullanılmıştır. ANCOR içinde kullanılan yöntem, çekici, itici ve etkisiz feromon değerlerinin tek yapı altında birleştirerek Montgomery'nin[18]'nin kullandığı yöntemden ayrılmıştır.

6.2.1 İlklendirme safhası

İlklendirme safhası, algoritmanın başlangıcıdır. Bu safhada, yuvada bulunan karıncaların besin bulmak için sahaya efektif bir şekilde yayılması sağlanmaktadır. Sahada daha önce hiç karınca bulunmadığı varsayılarak, algılayıcı ağında bulunan tüm düğümlerin feromon değeri ”1”, yani etkisiz feromon yapılır. Bir karınca, bir sonraki adımında hangi düğüme gideceğini, bulunduğu düğümün komşularının feromon değerine göre belirler. Bir sonraki adımda, komşu düğümler arasında en yüksek feromon değerine sahip olan düğüme gidilir. Eğer karıncanın komşularında en yüksek feromon değerine birden fazla düğüm sahip ise, bir sonraki adımı, maksimum feromona sahip düğümler arasından rastgele seçilir. Yuvadan ayrılacak olan ilk karınca bir sonraki düğümü rastgele seçmek zorundadır. Başlangıçta, yuvanın etrafında bulunan düğümlerin hepsinin feromon değeri eşittir. Bir karınca yeni bir düğüme geldiğinde, düğümün hedef olup olmadığını kontrol eder. Hedef değil ise düğümün feromon değerini günceller ve bir sonra gideceği düğümü seçer. Feromon değeri güncellenen düğüm, yeni feromon değerini komşularına bildirir. Böylece, düğüme komşu olan düğümlerin feromon listeleri de güncellenmiş olur. İlklendirme safhasındaki feromon güncelleme işlemi, karıncaların sahaya yayılmalarını sağlamak açısından önemlidir. Bir karınca düğüme geldiğinde, düğüm hedef değil ise düğümün feromon değerini itici olarak işaretleyerek, kendisinden sonra gelen karıncaların söz konusu düğüme uğramamalarını sağlar. İlklendirme safhası, karıncalardan birisi hedefi bulana kadar sürer.

6.2.2 Güçlendirme safhası

İlklendirme safhası karıncalardan birisi hedefi bulduğunda sona erer. Hedef bulunduğunda, hedeften yuvaya doğru bir yolun oluşturulması gerekmektedir. Algoritmanın güçlendirme safhasında amaç, yuva ile hedef düğüm arasında bir yol oluşturmaktır. Karıncaları hedefi bulan karıncanın izlediği yola çekmek için yüksek feromon değerinin çekiciliği kullanılır. Güçlendirilmiş yol üzerinde bulunan düğümlerin feromon değerlerinin etkisiz feromon değerinden yüksek olması beklenir. Hedefi bulan karınca yuvaya geri dönüş yolunda, geçtiği her düğüme maksimum feromon bırakarak, diğer karıncaların kendi izinden gelmesini sağlar. Yuvaya doğru olan yolu izlemek için sürekli olarak komşuların feromon değerleri arasında en yüksek feromon değerine

sahip olan düğüme gidilmesi gerekmektedir. İtici feromon, yuvaya dönüş yolunun bulunabilmesi için gerekmektedir. Denklem 6.4'de bulunan $maxHop - hopCount$ ifadesi, yuvaya dönüş yolunun buharlaştırma işleminden etkilenmemesi açısından önemlidir.

6.2.3 Yönlendirme safhası

Çekici feromon değerini tanımladıktan sonra, yönlendirme (veri paketi aktarımı) üç adımda gerçekleştirilmektedir.

1. Herhangi bir düğümde, düğümün komşularının feromon değerlerini kontrol et.
2. Bir sonraki adım, komşular arasındaki en yüksek feromon değerine sahip olan düğümler arasından seç.
3. Üzerinde bulunduğu düğümün feromon değerini yenile.

Karıncalar, hedefi bulan karınca hariç olmak üzere, algoritmanın yukarıdaki üç adımını sonsuz döngü içinde gerçekleştirirler. Her düğümde, karınca bir sonraki adımda gideceği düğümü aşağıdaki denklemden ortaya çıkacak olasılık değerlerine göre belirler.

$$DüğümOlasılığı = (\beta \times düğüm_i feromon) + ((1 - \beta) \times rastgeleSayı) \tag{6.3}$$

β değeri, karıncaların farklı hedeflere ulaşabilmesi için, en çok kullanılan yol dışına çıkabilmelerini sağlar. Karınca, formüle göre seçim olasılığını komşu düğümler için hesaplar. Komşular arasında en yüksek olasılığa sahip olan düğümü bir sonraki adım için seçer. $\beta > 0.5$ için algoritma en çok kullanılan yolu seçer. $\beta < 0.5$ için karıncaların yoldan ayrılıp farklı kaynakları aramaları çoğalır. Bir düğümde feromon değeri artışı:

$$yeniFeromon = \left[(maxHop - hopCount) \times feromon\right]^{\alpha} \times eskiFeromon \tag{6.4}$$

maxHop, bir karıncanın ağ üzerinde yok olmadan dolaşabileceği en uzun süreyi belirtir. hopCount, karıncanın şimdiye kadar yapmış olduğu düğüm atlama sayısıdır. Feromon, karınca tarafından salgılanan feromon değeridir. eskiFeromon, düğümün eski feromon değeridir. yeniFeromon, düğümün yeni feromon değeridir. α, aşağıdaki denklem 6.5'e göre pozitif-negatif feromon değerini verir:

$$\alpha = \left\{ \begin{array}{cc} 1 & ph < 1 \\ -1 & ph \geq 1 \end{array} \right\} \tag{6.5}$$

Alıcıda bulunan kullanıcı bir sorgu yaptığında, sorgulanan düğümler tüm ağ üzerinde aranacaktır. Yeni bir sorgu yapıldığında, yuvada yeni karıncalar yaratılıp ağa gönderilecektir. Bu durumda yuva yakınlarında feromon değeri yüksek olacaktır. İlklendirme safhasında alıcı çevresi itici feromon ile kaplanır. Yönlendirme safhasında, itici feromon normal feromon ile değiştirilmiş olacaktır. Karıncalar ağa yuvadan çıktıkları için yuva etrafında feromon çok yoğundur. Karıncalar yuvadan uzaklaştıkça feromon yoğunluğu azalacaktır. Karıncalardan biri hedefi bulup yuvaya geri döndüğünde, yönlendirme safhası başlar. Yönlendirme safhasında karınca hedefi ya da yuvayı, işaretli düğümleri izleyerek kolaylıkla bulacaktır. Bütün safhalarda, çok kullanılan yol ile rastgele seçilen yol arasında ödün vermek gerekmektedir. Rastgele seçimin etkisini arttıran β değeri, uygulamaya özel bir değer olarak verilebilir. Örneğin, hareketli bir aracın takibinde rastgele aramanın yüksek olması, hareketli aracın yeni konumunu belirlemek açısından faydalı olacaktır. Uygulamamızda, hedef düğümler zamanla değişecektir. Bu durumla başa çıkabilmek için rastgele aramayı uyarlanabilir olarak değiştirmek gerekmektedir. Bu değişim denklem 6.6 ile gerçekleştirilir:

$$olasılık = \big(\beta \times düğüm_i ph\big) + \big(1 - \beta\big) \times \frac{rastgeleSayı}{maxHop - hopCount} \tag{6.6}$$

7 PARALEL ALGORİTMA: PANCOR

Seri algoritma küçük modellerde (625 düğüm ve 40 karınca) efektif çalışmakla birlikte, daha büyük ağlarda ve daha yüksek sayıda karınca bulunan modelleri (2500 düğüm ve 4096 karınca) gerçekleyememektedir.

Seri algoritmada, hem karıncalar hem de düğümler için aynı adres alanı kullanılmaktadır. Bu durum, karınca ve düğüm sayısının, bir uygulamanın kapsayabileceği hafıza alanına sığmasını zorunlu kılmaktadır. Buradaki kısıtlamayı ortadan kaldırabilmek için, feromon haritası ile karıncalar farklı süreçlerde çalışabilecek şekilde parçalanmıştır. Böylece, ana işlemde sadece feromon haritası ile ilgili işlemler kalmıştır.

Düğümlerle ilgili işlemler, tanımlamış olan düğüm nesnesi içinde gerçeklenmiştir. Seri algoritmadan farklı olarak, düğümlerin komşularının feromon değerleri değil, düğüm numaraları saklanmıştır. Böylece, bir düğüm için feromon güncellemesi yapıldığında, yeni değerin diğer düğümlere yansıtılması gerekmemektedir. Karıncaların hareketi, seri algoritmada anlatıldığı gibi modellenerek her bir karıncanın bir thread içinde çalışması sağlanmıştır. Böylece örneğin 11 işlemcili bir sistemde feromon haritasını merkez tutmakta, geri kalan 10 sistem ise, kendilerinde istenen miktarda karınca threadi yaratabilmektedir.

Karınca ile feromon haritasının ilişkisi, merkez ve köleler arasında yapılan mesajlaşmalarla sağlanmaktadır. Karıncaların çalıştığı sistemlerde, düğüm matrisi bulunmamaktadır. Karınca için gerekli olan düğüm bilgisi, düğümlerin saklandığı merkezde bulunmaktadır. Karınca, merkeze, kendisine gerekli olan düğüm numarasını bildirir. Merkez, ilgili düğüm nesnesini, bir mesaj içinde karıncaya gönderir. Karınca düğüm üzerinde değişiklik yapmak istediğinde, değişiklik istemi mesajla merkeze bildirilmektedir. Merkeze ulaşan değişiklik istemi, ilgili düğüm için yapılmaktadır. Böylece, bütün sistem ortak bir düğüm matrisini kullanmaktadır.

7.1 Paralel Uygulama Detayları

Paralel yönlendirme uygulaması, temel olarak iki kısımdan oluşur. Merkezde çalışan kısım, kölelerde çalışan kısım:

7.1.1 Merkezde çalışan kısım

Paralel sistem çalıştırıldığında, bütün simülasyonu yöneten merkezi süreç, öncelikle sistemde kaç işlemci bulunduğunu ve her bir işlemci için kaç adet karınca yaratılacağını saptar. Sistemde kaç düğüm bulunması isteniyorsa, o kadar sayıda düğüm yaratılır. Yaratılan düğümlerin birbirlerine olan komşulukları hesaplanır.

Sistemde köle süreçler de çalışmaya başladığından, simülasyonun bilgilerinin hazırlanmasını beklemelerini sağlamak amacıyla mpi.barrier() kullanılır. Mpi.barrier(), paralel sistemde bulunan tüm süreçlerin, uygulama içinde aynı noktaya gelmelerini sağlar. Komşuluk hesaplaması yapıldıktan sonra, kölelere, karıncalarını yaratmaları için mesaj gönderilir. Yaratılan karıncalar, hemen düğümlerde gezmeye başlamamaları için, ayrıca kendilerine gönderilecek olan başlama mesajını beklerler.

Tüm köle süreçlerde karınca threadleri yaratıldıktan sonra, köle süreçlere, karıncaların düğümlerde dolaşmalarını başlatacak olan simülasyonu başlat emri gönderilir. Bu aşamadan sonra, merkez uygulama, sahada gezinen karıncalar için gerekecek düğüm bilgilerinin karıncalara aktarılmasını sağlayan bir sonsuz döngü içine girer.

Gelen isteklerin sınıflandırmasının kolay yapılabilmesi için, her bir işlem grubuna ayrı bir mesaj etiketi verilmiştir. Seri algoritma içinde aynı hafıza alanında bulunan karıncalar ve düğümler arasında bu etkileşim kolaylıkla sağlanmaktadır.

Paralel yöntemde, düğümler ve karıncalar farklı hafıza alanlarındadır. Bu nedenle merkezi sürecin hafıza alanında bulunan düğümlerin bilgilerinin güncellenmesi işlemi, karıncalar tarafından, güncellenmesi gereken bilginin, ya da güncelleme için kullanılacak olan verinin merkez sürece gönderilerek, merkez süreç tarafından düğüme uygulanması ile gerçeklenmiştir. Aşağıda, uygulamada kullanılan mesaj etiketleri ve açıklamaları bulunmaktadır:

GETMINPH : Düğümün komşularına ait en küçük feromon değerine sahip olan düğüm numarası isteği. Karınca tarafından merkeze gönderilir. Mesaj içinde, karıncanın üzerinde bulunduğu düğümün numarası ve bir önceki adımda üzerinde bulunduğu düğümün numarası bulunur.

SENDMINPH : En düşük feromon değerine sahip olan düğüm numarasını, GETMINPH etiketli mesajla isteyen karıncaya gönderir. Merkez tarafından karıncaya gönderilir. Mesaj içinde karıncanın üzerinde bulunduğu düğümün komşuları içinde en düşük feromon değerine sahip olan düğümün numarası bulunur.

GETMAXPH : Düğümün komşularına ait en yüksek feromon değerine sahip olan düğüm numarası isteği. Karınca tarafından merkeze gönderilir. Mesaj içinde, karıncanın üzerinde bulunduğu düğümün numarası bulunur.

SENDMAXPH : En yüksek feromon değerine sahip olan düğüm numarasını, GETMAXPH etiketli mesajla isteyen karıncaya gönderir. Merkez tarafından karıncaya gönderilir. Mesaj içinde karıncanın üzerinde bulunduğu düğümün komşuları içinde en yüksek feromon değerine sahip olan düğümün numarası bulunur.

GETPROBNEXTNODE : Karıncanın bir sonraki adımda hangi komşu düğüme gideceğini hesaplaması için merkez gönderdiği mesaj. Karıncadan merkeze gönderilir. Mesaj içinde karıncanın terk etmek üzere olduğu düğümün numarası bulunur.

SENDPROBNEXTNODE : Merkezin düğüm matrisini kullanarak istek yapan karıncanın bir sonraki adımda hangi düğüme gideceğini karıncaya bildirirken kullandığı mesaj etiketi. Merkezden karıncaya gönderilir. Karıncanın bir sonraki adımda hangi düğüme gideceği bilgisi gönderilir.

GETPH : Karıncanın, bir düğümün feromon değerini öğrenmek istediğinde merkeze gönderdiği mesaj etiketi. Karıncadan merkeze gönderilir. Mesaj içinde, hangi düğümün feromon değerinin istendiği bulunur.

SENDPH : Merkezin, bir düğümün feromon değerini ilgili karıncaya gönderirken kullandığı mesaj etiketi. Merkezden karıncaya gönderilir. Mesaj içinde, karınca tarafından istenen düğümün feromon değeri bulunur.

GETDATA : Karıncanın, bir düğümün hedef olup olmadığını belirleyen bilgisini merkezden sorarken kullandığı etiket. Karıncadan merkeze gönderilir. Mesaj içinde, bilgisi istenen düğümün numarası bulunur.

SENDDATA : Merkezin, ilgili karıncaya düğümün hedef bilgisini gönderirken kullandığı etiket. Merkezden karıncaya gönderilir. Mesaj içinde, düğümün hedef bilgisi bulunur.

UPDATEPH : Karıncanın düğümün feromon değerini güncellemek için gönderdiği etiket. Karıncadan merkeze gönderilir. Mesaj içinde, feromon değeri güncellenecek olan düğümün numarası ve güncellemede kullanılacak olan, karıncanın adım sayısı bilgisi vardır.

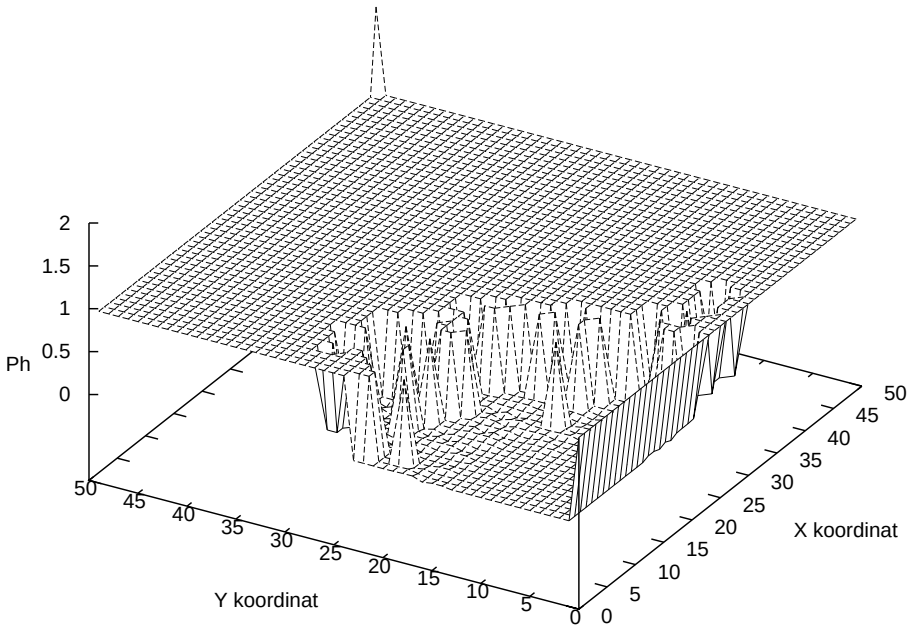
SETPH : Karıncanın, bir düğümün feromon değerini ataması için kullanılır. Karıncadan merkeze gönderilir. Mesaj içinde, feromon değeri değiştirilecek olan düğüm numarası ve yeni feromon değeri bulunur.

GETNODE : Karıncanın, yeni bir düğüme geldiğinde, ilgili düğüm nesnesini istemek için kullandığı etiket. Karıncadan merkeze gönderilir. Mesaj içinde, isteği yapan karınca threadinin numarası ve bilgisi istenen düğümün numarası bulunur.

SENDNODE : Merkez, kendisinden istenen düğüm bilgisini, ilgili köle sunucuya gönderir. Merkezden karıncaya gönderilir. Mesaj içinde düğüm nesnesi bulunur.

FINALIZE : Simülasyonu sonlandırmak için gönderilir.

Merkezde çalışan uygulama, yukarıdaki mesajları bekleyip, gelen mesajın içeriğine göre ilgili sürece gereken bilgileri gönderir. Burada önemli olan nokta, seri algoritmada aynı hafıza alanında bulunan bilgilerin, paralel uygulamada farklı hafıza alanlarına taşınabilmiş olmasıdır. Böylece, sisteme eklenebilecek olan karınca sayısı, sisteme

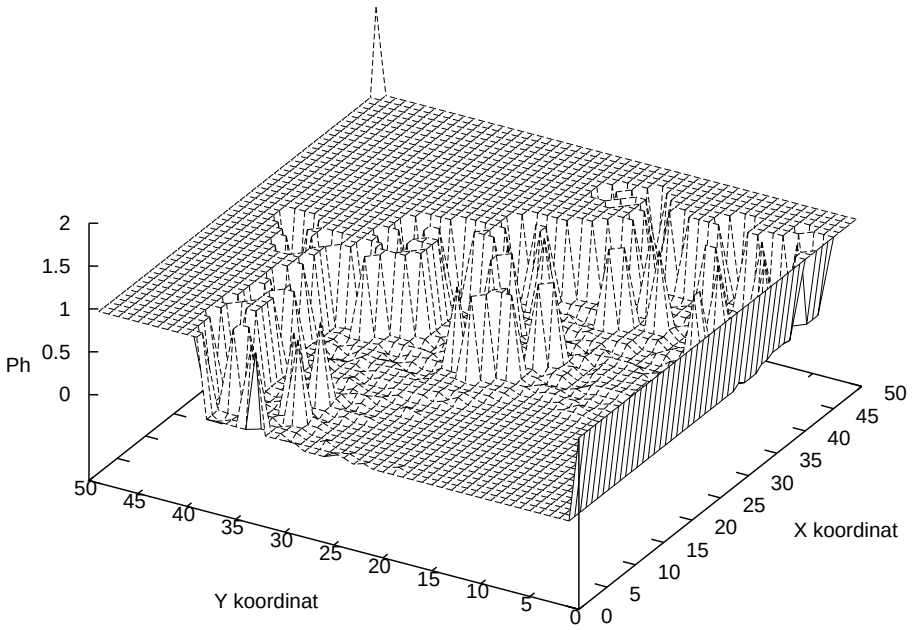


Şekil 7.1: 16896 mesaj sonrasında feromon yoğunluk haritası

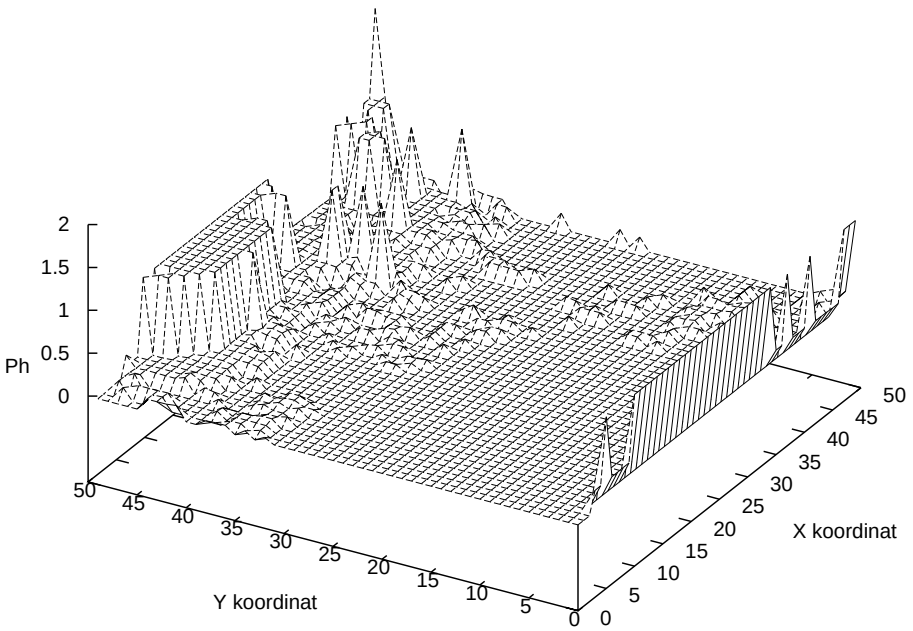
eklenebilecek hesaplama sunucu sayısına göre lineer artabilecektir. Burada sınırlanan diğer veri ise, merkez süreç tarafından güncellenen düğüm bilgileridir. Düğüm bilgilerinin de karıncalar gibi dağıtılabileceği göz önüne alındığında, seri algoritmanın hem hesaplama ihtiyacı, hem de hafıza kısıtları nedeniyle ulaşamayacağı simülasyon büyüklüklerine kolaylıkla ulaşılabilecektir. Testlerde, 50x50 düğüm matrisi üzerinde 4096 karınca içeren simülasyonlar başarıyla koşturulmuştur. Test sırasında kullanılan sistem, 16 işlemcili 64 GB hafızaya sahip bir SMP sistemdir. Masaüstü sistemlerde yapılan denemelerde, işletim sisteminin ayarlamalarına göre, bir süreç içinde 256'ya kadar thread kolaylıkla çalıştırılabilmiştir. Bu durumda, sisteme eklenecek her bir işlemci için, ek 256 karınca daha koşturulabilecektir. 512 çekirdekli dağıtık bir sistemde, 131.072 karınca threadi koşturmak mümkün olabilecektir.

7.1.2 Kölelerde çalışan kısım

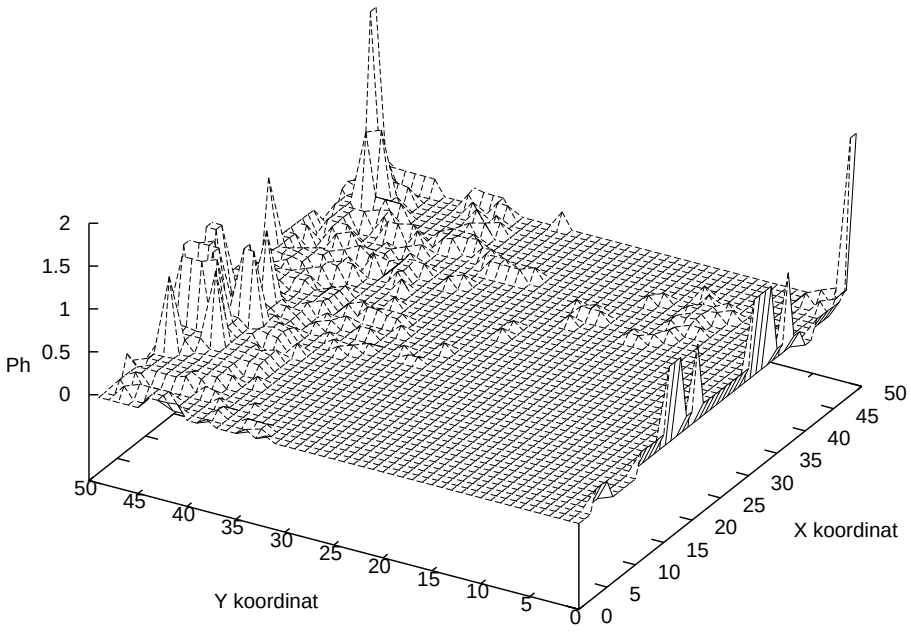
Köleler, öncelikle merkez sürecin simülasyon için gereken ortamı hazırlamasını beklerler. Bekletme işlemi, mpi.barrier() ile sağlanır. Merkez, bitirilmesi gereken adımlardan sonra gelen mpi.barrier() komutuna vardığında, köleler de bir sonraki



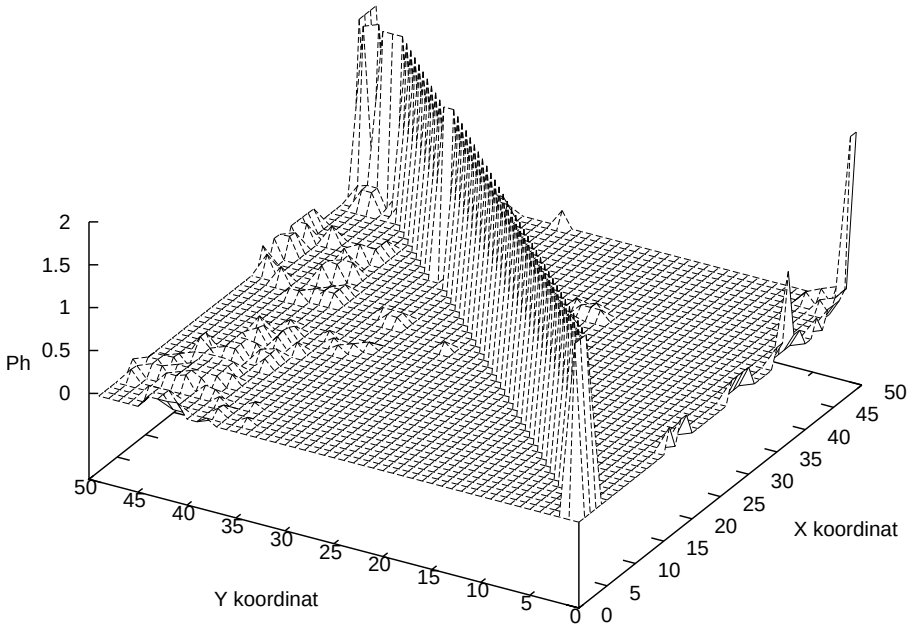
Şekil 7.2: 43392 mesaj sonrasında feromon yoğunluk haritası



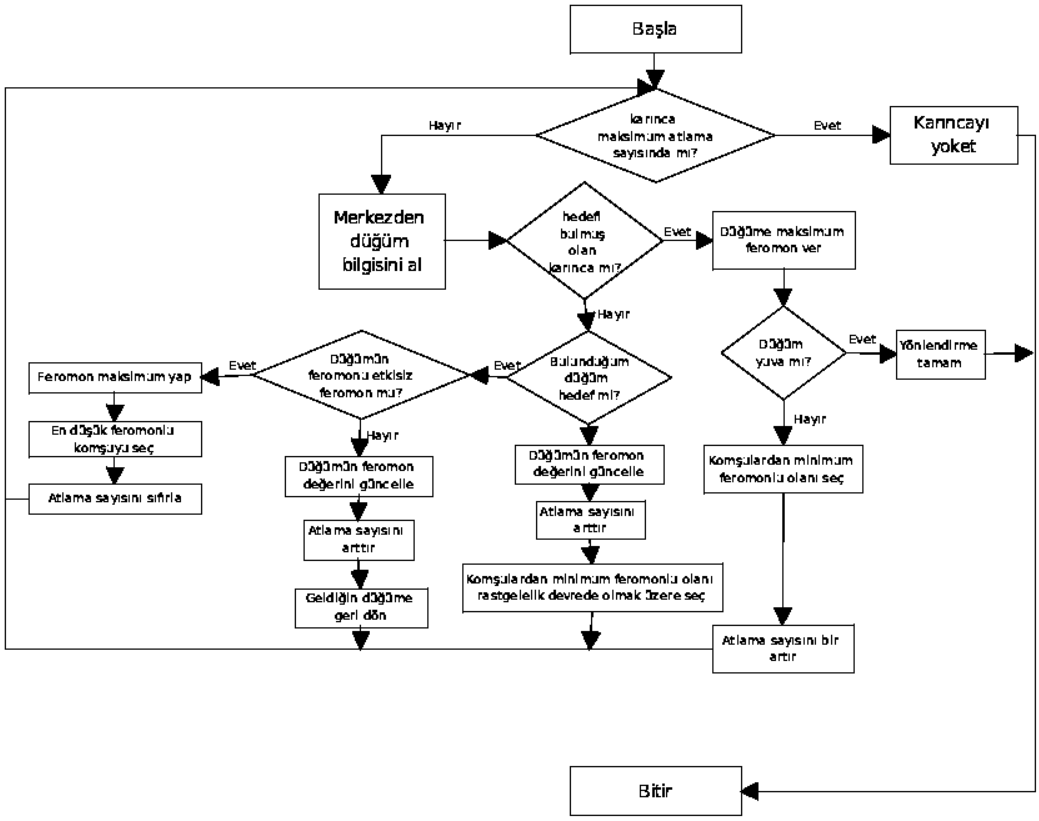
Şekil 7.3: 88704 mesaj sonrasında feromon yoğunluk haritası



Şekil 7.4: 99072 mesaj sonrasında feromon yoğunluk haritası



Şekil 7.5: 115968 mesaj sonrasında feromon yoğunluk haritası



Şekil 7.6: Karınca hareketi akış şeması

adıma geçebilmektedirler. Merkez, düğümleri yaratıp komşuluklarını hesapladıktan sonra, köleler karınca threadlerini yaratmaya başlarlar. Yaratılan karınca threadleri, merkezden tüm kölelerin karınca yaratma işlemini bitirdikleri bilgisini verene kadar, düğümlerde dolaşmaya başlamazlar. Tüm karıncalar yaratılınca, sistemin tamamına başlama mesajı gönderilir. Bu aşamadan sonra, her bir karınca threadi kendi başına hareket etmeye başlar. Karınca threadi ana gövdesi, simülasyonun gerçekleştiği bölgedir. Bu gövde içinde karıncaların düğümler ile ilgili işlemler yapması ya da merkeze yaptırması sağlanır. Simülasyon başlatıldığında, karınca merkezden üzerinde bulunduğu düğümün bilgilerini ister. Merkez, karıncaya istediği düğümü nesne olarak gönderir. Şekil 7.6'da , karıncanın düğüm matrisinde hareket ederken kullandığı algoritma bulunmaktadır.

Seri algoritmada karınca nesnesinin içinde yapılan düğüm ile ilgili işlemler, artık karınca tarafından gerçekleştirilemezler. Bunun yerine karıncalar, merkeze yapmak

istedikleri işlemi, işlem için gereken bilgilerle beraber bildirirler. Düğüm bilgilerinin bulunduğu merkez uygulaması, karıncalardan gelen mesajları alarak, istenen işlemi gerçekleştirir. Merkez uygulamadaki mesajların cevaplandırılması işlemi, düğüm matrisinde değişikliklerin atomik olmasını sağlamaktadır.

8 SONUÇLAR VE ÖNERİLER

Bir düğümde aynı anda işlem görebilecek karınca sayısı sınırlanmadığı zaman, karıncaların dönüş yolunu bulmak için kullandıkları feromon haritası düzgün oluşturulamamaktadır. Bu durumu engellemek için, simülasyon sırasında bir düğümde aynı anda bulunabilecek karınca sayısı sınırlandırılmıştır. Şekil 8.1'de paralel algoritmalar, seri algoritmadan daha iyi performans göstermişlerdir. Sınırlandırma olmadan çalışan paralel algoritmada karıncalar başlangıçtaki düğümlerden, karıncaların düğümün feromon değerlerini değiştiremeden geçebilmektedirler. Bu duruma, merkezi yöneticinin her bir karıncaya pek çok mesajla hizmet etmesi neden olmaktadır. Bir karınca düğüm bilgisini alıp işlemlerini yaparak bir sonraki adımını seçene kadar, düğümden çok sayıda karınca geçmiş bulunmaktadır. Bu durum, karıncaların düğümlere yayılmasını da etkilemektedir.

Sınırlandırma işlemi, bir düğümde aynı anda bulunabilecek karınca sayısını belirterek yapılmaktadır. Düğüme gelen karınca, öncelikle düğümün üzerinde bulunan karınca sayısını kontrol eder. Karınca sayısı belirtilen değerden az ise, karınca düğüm üzerinde işlemlerine devam edebilir. Karınca sayısı düğüm için verilen maksimum değere ulaşmışsa, karınca düğüm üzerindeki karınca sayısı azalana kadar beklemeye geçer. Bekleme işlemi, karıncanın düğüm durumunu merkeze sürekli olarak sorması ile gerçeklenmiştir. Testlerde, kısıtlamalı model için bir düğümde en fazla 8 karıncaya izin verilmiştir.

Seri algoritma, ağ üzerindeki bütün karıncaları sırayla hareket ettirmektedir. Bu durum, doğal ortama tam olarak uymamaktadır. Paralel uygulama ile, karıncalar kendi başlarına hareket edebilmektedirler. Seri uygulamada simülasyonun ilk adımında tüm karıncalar yuvadan ayrılmışken, paralel algoritmada karıncaların yuvayı aynı anda terketmesi mümkün olmamaktadır. Bu durum, doğal ortama daha yakın bir davranış şeklinin ortaya çıkmasını sağlamaktadır.

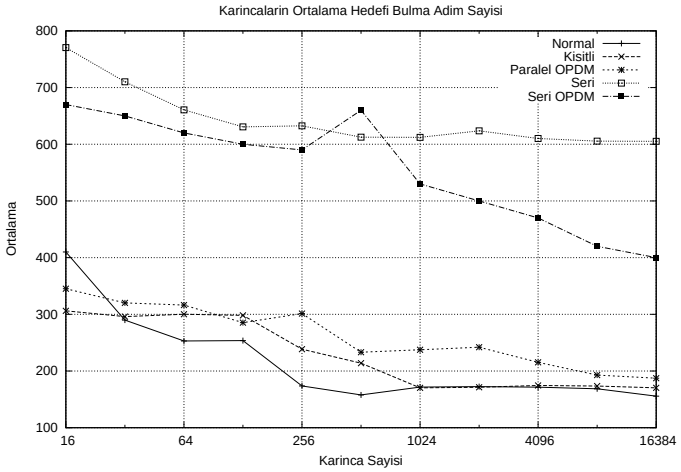
Paralel algoritmada karıncalar kısıtlamasız hareket ettiklerinden, bir düğümde bir karınca işlemini bitiremeden, çok sayıda diğer karınca aynı düğümden

geçebilmektedir. Paralel algoritmada merkezdeki yönetici kısım, engelleyici mesaj alış veriş komutları kullanır. Çok sayıda karınca kullanılan simülasyonlarda, karıncaların merkezle yaptıkları haberleşmede yüksek sayıda mesaj kullanıldığından, gönderilen mesajlar tampon bölgede sıraya girmek zorunda kalırlar. Bir karınca bir düğümde işlerini bitirene kadar, düğümün özelliklerine bağlı olarak en az üç mesaj göndermek zorundadır. Yuvadan ayrılan karıncaların tamamı merkeze mesaj gönderirler. Merkez, tüm mesajları işlerken, gelmekte olan mesajları da tampon belleğinde beklemeye alır. Bu durum, karıncaların düğümler yeni değerlerini alamadan düğümleri geçmelerine sebep olur. Sınırlandırma uygulanmadığında karıncaların hedefi bulma performansları seri algoritmaya göre Şekil 8.1'de görüldüğü gibi daha iyidir. Ancak sınırlamasız karıncaların oluşturduğu feromon haritasını kullanan karıncaların yuvaya dönüş performansları seri algoritmaya göre Şekil 8.2'de görüldüğü gibi çok kötü gerçekleşmektedir.

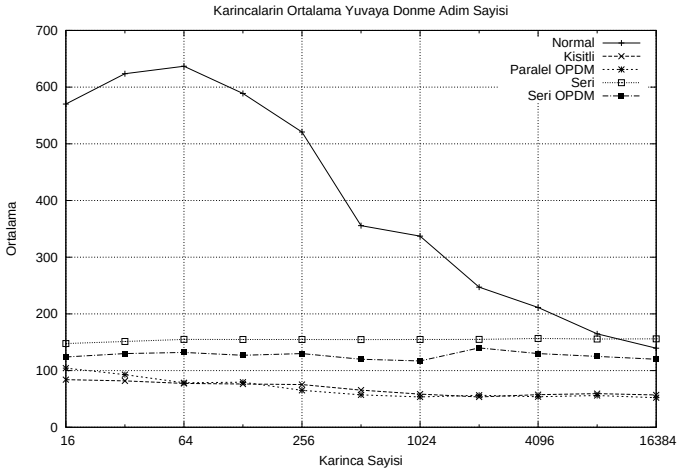
Seri algoritmanın ilk gerçeklemesinde, düğümlere gelen karıncaların bıraktıkları feromon komşu düğümlere etki etmiyordu. Gerçek hayatta, bir düğümde çekici ya da itici feromon bulunması durumunda, komşu düğümleri etkilemektedir. Komşu düğümdeki feromon miktarı, bir düğümün feromon miktarını değiştirebilir. Demiray ve Altılar ANCOR algoritmasını daha da geliştirdikleri makalelerinde[3] bu durumu gerçeklemişlerdir. Bu yeni modele optimal feromon dağıtım modeli (optimal pheromone distribution model, OPDM) adı verilmiştir. Paralel algoritmada OPDM modeli de kullanılarak yapılan hedefi bulma testlerinde sınırlandırılmış modelle paralel OPDM modeli benzer sonuçlar vermiştir. Seri OPDM ve seri modeller, hedefi bulma testinde paralel modellerin tamamının gerisinde kalmıştır.

Hedeften yuvaya dönüş rakamlarında, kısıtlama olmayan paralel model ancak çok yüksek karınca sayılarında yapılan testlerde diğer modellere yaklaşabilmiştir. Ancak bu modelde haritayı oluştururken karıncaların düğümlere etkisi tam yansıtılamaz. Kısıtlama yapılan paralel model ve paralel OPDM modeli, seri algoritmadan daha iyi performans sergilemişlerdir.

Paralel algoritmalar, gerçek hayatta biribirinden bağımsız hareket eden canlıları örnek alan algoritma modellerinin seri algoritmalardan daha iyi modellenebilmesini



Şekil 8.1: Karıncaların hedefi bulma adım ortalamaları



Şekil 8.2: Karıncaların yuvaya dönme adım ortalamaları

sağlarlar. Paralel algoritmaların kullanılmasıyla, çok daha fazla sayıda karınca içeren modeller çok daha büyük düğüm sayılarında çalıştırılabilirler. Şekillerde de görüldüğü gibi, paralel algoritmalar seri algoritmalardan daha iyi performans sergilemişlerdir. Doğal hayatı model alan algoritmaların düzgün modellenebilmesi, modele örnek olan canlıların gerçek hayattaki davranış şekillerinin algoritmaya ne kadar iyi aktarılabildiğine bağlıdır. Bir topluluğun davranış şekillerini kullanan modellerde paralel algoritmalar kullanmak, topluluğu daha rahat algoritmaya aktarabilmeyi ve daha doğru sonuçlara ulaşabilmeyi de sağlar.

KAYNAKLAR

[1] Prasanna V., Yu Y., Krishnamachari B.,"Information Processing and Routing in Wireless Sensor Networks", ***World Scientific Publishing Co.***, (2006).

[2] Demiray D., Altilar D.T., "ANCOR: A Novel Ant Colony Routing Approach for Sensor Networks" ***Int. J. IT&IC Volume 1 Issue 4***, (2007).

[3] Demiray D., Altilar D.T., "Optimal Pheromone Distribution for ANCOR", ***IEEE Swarm Intelligence Symposium***, (2008).

[4] Dorigo M., Maniezzo V., Colorni A.,"Ant System: Optimization by a Colony of Cooperating Agents", ***IEEE Transactions On Systems, Man And Cybernetics-Part B: Cybernetics***, Vol 26, No 1, (1996).

[5] Bello R., Nowe A., Cabellero Y., Gomez Y., Vrancx P. "A Model Based On Ant Colony and Rough Set Theory To Feature Selection", ***GECCO'05***, (2005).

[6] Parpinelli R.S., Lopes H.S., Freitas A.A., "Data Mining With an Ant Colony Optimization Algorithm", ***IEEE Transactions On Evolutionary Computation***, Vol 6, No 4, (2002).

[7] Das S., Singh G., Pujar G., Koduru P.,"Ant Colony Algorithms for Routing in Sensor Networks", ***GECCO 2004***, LBP053, (2004).

[8] Akyıldız I.F. , Sankarasubramaniam Y., Çayırcı E., "Wireless Sensor Networks: a survey", ***Computer Networks, Elsevier Science B.V.***, 38, 393-422,(2002).

[9] Wilkinson B., Allen M. "Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers", ***Prentice Hall***, 3-26, 38-46, 404-410, (1999).

[10] Flynn M.J., "Very High Speed Computing Systems", ***Proceedings of the IEEE***, 54(12), 1901-1909, (1966).

[11] Dally W.J., Seitz C.L., "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks", ***IEEE Trans. Comp.*** , Volume 36, Num:5, 547-553, (1987).

[12] Braam P.J., "The Lustre Storage Architecture", ***White Paper, Cluster File Systems Inc.***, Vol 23, (2003).

[13] Ross R, Thakur R., Walter H., Carns P., "PVFS: A Parallel File System For Linux Clusters", ***Proceedings of the 4th Annual Linux Showcase and Conference***, 317-323, (2000).

[14] 2008, Open MPI: Open Source High Performance Computing, http://www.openmpi.org , (***Ziyaret tarihi: 28 Ekim 2008***).

[15] Mallon E.B., Pratt S.C., Franks N.R., ”Individual and Collective Decision Making During Nest Site Selection by The Ant Leptothorax Albipennis”, ***Behavioral Ecol. Sociobiol.,Springer Verlag***, 50, 352-359, (2001).

[16] Zhang Y., Kuhn L.D., Fromherz M.P.J., ”Improvements on ant routing for sensor networks”, ***ANTS 2004,LNCS 3172, Springer-Verlag***, 154-165, (2004).

[17] Di Caro G., Dorigo M., ”Antnet: Distributed Stigmergetic Control for Communication Networks”, ***Journal of Artificial Research***, 9, 317-365, (1998).

[18] Montgomery J., Randall M., ”Anti-pheromone as a Tool for Better Exploration of Search Space, Dorigo M. et al. (Eds.)”, ***ANTS 2002, LNCS 2463, Springer-Verlag***, 100-110, (2002).

EKLER

```
import mpi
import math
import time
import threading
import sys
import random
from datetime import datetime
import string

global SINK
RADIO_RANGE = 1.8
NULL_PH     = 1.0
MAX_PH      = 2.0
MIN_PH      = 0.0
EVAP_CONST  = 0.001
ANT_PH      = 1.0
MAX_HOP     = 1000
TARGET_DATA = 10000000001
SINK        = 0
BETA        = 0.75
global Nodes
total_ants  = 1024
matris = []

class tags:
  """
  Master ile karıncalar arasındaki haberleşme
  trafiginin düzenlenmesi icin, her bir mesaj
  grubunun işaretlenmesi gerekiyor.

  Aşağıdaki değerler, MPI için mesaj tag
  değerleridir
  """
  def __init__(self):
    self.GETMINPH  = 100
    self.SENDMINPH = 101
    self.GETMAXPH  = 102
    self.SENDMAXPH = 103
    self.GETPROBNEXTNODE  = 105
    self.SENDPROBNEXTNODE = 106
    self.GETPH     = 107
    self.SENDPH    = 108
    self.GETDATA   = 109
    self.SENDDATA  = 110
    self.UPDATEPH  = 111
    self.SETPH     = 112
    self.GETNODE   = 113
    self.SENDNODE  = 114
    self.FINALIZE  = 115

mtag = tags()

def getNode(antid,idx):
  """
  Bir karınca, sonraki adımda nereye gideceğini bulabilmek
  için, üzerinde bulunduğu node ile ilgili bilgilere ihtiyaç
  duyar.

  Mesajlaşma sayısını azaltabilmek için, karar verme öncesinde
  ilgili node için tüm bilgi alınıyor.
  """
  mpi.send([antid,idx],0,tag = mtag.GETNODE)
  xx ,stat  = mpi.recv(0, tag = mtag.SENDNODE)
  return xx

class Ant(threading.Thread):
  """
  karinca sinifi, slave makinalar icin lokal olacak
  """
```

```
def __init__(self,_id,_node_id):
  _name = "Ant%d" % _id
  threading.Thread.__init__(self, name = _name)
  self.idx = _id
  self.node_id = _node_id
  self.previous_node_id = -1
  self.next_node_id = -1
  self.hop = 0
  self.new_found_ant = False
  self.startSim = False


def updatePh(self,node_id,hopCount):
  """
  Üzerinde bulunulan node için değer atama işlemleri,
  lokalde değil, merkezde yapılmak zorundadır.

  update işlemi için gereken bilgi merkeze gönderilir.
  update işlemini master proses yapar.
  """
  temp = [node_id,hopCount]
  mpi.send(temp,0,tag = mtag.UPDATEPH)

def setPh(self,node_id,newPh):
  """
  merkeze, id'si verilen node icin yeni ph degeri
  atanmasi icin mesaj gonderir.

  iki elemanli liste gonderilir. birinci eleman
  node_id, ikinci eleman yeni ph degeri.
  """
  mpi.send((node_id,newPh),0,tag = mtag.SETPH)

def getPh(self,xx):
  mpi.send(xx, 0, tag = mtag.GETPH)
  ph,stat = mpi.recv(0, tag = mtag.SENDPH)
  return ph

def getData(self,xx):
  mpi.send(xx, 0, tag = mtag.GETDATA)
  temp,stat = mpi.recv(tag = mtag.SENDDATA)
  return 0

def minPhNode(self,node_id):
  temp_id = -1
  temp = [node_id,self.previous_node_id]
  mpi.send(temp,0,tag = mtag.GETMINPH)
  temp_id,stat = mpi.recv(tag = mtag.SENDMINPH)
  return temp_id

def maxPhNode(self,node_id):
  temp_id = []
  mpi.send(node_id, 0 ,tag = mtag.GETMAXPH)
  temp_id,stat = mpi.recv( tag = mtag.SENDMAXPH)
  return temp_id


def probableNextNode(self,node_id):
  temp_id = -1
  temp = [self.node_id,node_id]
  mpi.send(temp,0,tag = mtag.GETPROBNEXTNODE)
  temp_id,stat = mpi.recv(tag = mtag.SENDPROBNEXTNODE)
  return temp_id


def run(self):
  time.sleep(0.01)
  while self.startSim == False:
    time.sleep(3)

  tempId = -1
  antid = -1
  while True:
    time.sleep(0.01)
    if self.hop == MAX_HOP:
```

```
        mpi.send(0,0,tag = mtag.FINALIZE)
        sys.exit()
      now = datetime.now()
      antid, xnode = getNode(self.idx,self.node_id)
      if self.new_found_ant == True:
        self.setPh(self.node_id,MAX_PH)
        if (self.node_id == SINK):
          print  "dondum ",self.idx,self.hop
          self.new_found_ant == False
          mpi.send(0,0,tag = mtag.FINALIZE)
          sys.exit()
        else:
          minId = self.minPhNode(self.node_id)
          self.previous_node_id = self.node_id
          self.node_id = minId
          self.hop += 1
      elif (xnode.data == TARGET_DATA):
        if xnode.ph == NULL_PH:
          print "buldum",self.idx,self.hop
          self.new_found_ant = True
          self.setPh(self.node_id,MAX_PH)
          minId = self.minPhNode(self.node_id)
          self.previous_node_id = self.node_id
          self.node_id = minId
          self.hop = 0
        else:
          self.node_id = tempId
          self.updatePh(self.node_id,self.hop)
          self.node_id = self.previous_node_id
          self.previous_node_id = tempId
          self.hop += 1
      else:
        if xnode.ph <= NULL_PH:
          self.updatePh(self.node_id,self.hop)
        self.previous_node_id = self.node_id
        self.node_id  = self.probableNextNode(self.node_id)
        self.hop += 1

def paint(nodelist,step):
  import numpy as N
  import pylab as pl
  def yukleMatris(nodelist):
    maxx = nodelist[0].x
    maxy = nodelist[0].y
    for nd in nodelist:
      if nd.x > maxx:
        maxx = nd.x
      if nd.y > maxy:
        maxy = nd.y

    _matris = []
    for j in range(int(maxy)):
      _matris.append([])
      for i in range(int(maxx)):
        _matris[len(_matris)-1].append([])
    for nd in nodelist:
      _matris[int(nd.x)-1][int(nd.y)-1]=nd.ph
    _matris[int(maxx)-1][int(maxy)-1] = 2.0
    return _matris

  def yukleDosya(fname):
    f = open(fname)
    satirlar = f.readlines()
    _matris = []
    for s in satirlar:
      _matris.append([])
      for z in string.split(s):
        _matris[len(matris)-1].append(float(z))
    return _matris


  def resimMatristen(nodelist,step):
    matris = yukleMatris(nodelist)
```

```
    im = pl.imshow(matris,interpolation = 'bilinear',\
    origin = 'lower',cmap = pl.cm.Spectral)
    pl.savefig("png/step%04d.png" % step)
    return 0

  def resimDosyadan(fname):
    matris = yukleDosya("plotdata/%s" % fname)
    im = pl.imshow(matris,interpolation = 'bilinear',\
    origin = 'lower',cmap = pl.cm.Spectral)
    pl.colorbar(im,boundaries=N.arange(0,2,0.1),spacing='uniform')
    pl.savefig("png/%s.png" % fname)
    return 0

  resimMatristen(nodelist,step)

class Node:
  def __init__(self,_id,_x,_y,_data):
    self.idx    = _id
    self.x      = _x * 1.0
    self.y      = _y * 1.0
    self.data   = _data
    self.ph     = NULL_PH
    self.neighbors = []
    self.alive = True


  def printNode(self):
    temp1  = "%d %f %f %f " % (self.idx, self.x, self.y, self.data )
    temp2 = ""
    for ngid in self.neighbors:
      temp2 += "%d " % ngid
    return temp1 + temp2 + "\n"

  def evaporate(self):
    #return 0
    if (self.ph > NULL_PH):
      self.ph -= EVAP_CONST
      if (self.ph < NULL_PH):
        self.ph = NULL_PH
    elif (self.ph < NULL_PH):
      self.ph += EVAP_CONST
      if (self.ph > NULL_PH):
        self.ph = NULL_PH
    else:
      self.ph = NULL_PH
    """
    for ng in self.neighbors:
      if (Nodes[ng].ph > NULL_PH):
        Nodes[ng].ph -= EVAP_CONST
        if (Nodes[ng].ph < NULL_PH):
          Nodes[ng].ph = NULL_PH
      elif (self.ph < NULL_PH):
        self.ph += EVAP_CONST
        if (self.ph > NULL_PH):
          self.ph = NULL_PH
      else:
        Nodes[ng].ph = NULL_PH
     """
  def setPH(self,newPH):
    self.ph = newPH

  def minPhNode(self, prevNode):
    ph = MAX_PH
    tempId = -1
    for node_id in self.neighbors:
      if Nodes[node_id].ph < ph:
        if prevNode != node_id:
          tempId = node_id
          ph = Nodes[node_id].ph
    return tempId
```

```
  def maxPhNode(self):
    ph = MIN_PH
    #tempId = -1
    for node_id in self.neighbors:
      if Nodes[node_id].ph > ph:
        if self.previous_node_id != node_id:
          tempId = node_id
          ph = Nodes[node_id].ph
    return (tempId,ph)

  def probableNextNode(self,ant_node_id):
    tempId = -1
    tempProb = -1
    for node_id in self.neighbors:
      if ant_node_id != node_id:
        randNo = random.randint(1,100) / 100.0
        nodeProb = (BETA * Nodes[node_id].ph) + ((1 - BETA) * randNo)
        if nodeProb > tempProb:
          tempProb = nodeProb
          tempId = node_id
    return tempId

  def computeBroadcast(self,nodeList):
    for node in nodeList:
      if (abs(node.x - self.x) > RADIO_RANGE):
        continue
      elif (abs(node.y - self.y) > RADIO_RANGE):
        continue
      elif node.idx != self.idx:
        xf1 = abs(node.x - self.x)
        yf1 = abs(node.y - self.y)
        xf  = xf1 * xf1
        yf  = yf1 * yf1
        dist = math.sqrt(xf + yf)
        if (dist <= RADIO_RANGE):
          self.neighbors.append(node.idx)

  def updatePh(self,antHop):
    temp = 0.0
    antHop = antHop * 1.0
    temp = ((MAX_HOP - antHop) / MAX_HOP) * ANT_PH
    if self.ph > NULL_PH:
      Nodes[self.idx].ph = Nodes[self.idx].ph  + temp
    else:
      Nodes[self.idx].ph = Nodes[self.idx].ph  - temp
    if (Nodes[self.idx].ph > MAX_PH):
      Nodes[self.idx].ph = MAX_PH
    elif (Nodes[self.idx].ph < MIN_PH):
      Nodes[self.idx].ph = MIN_PH


class neighborhood:
  def __init__(self,_idx,_start,_finish):
    _name="neighbor-%d" % _idx
    self.name = _name
    self.startid  = _start
    self.finishid = _finish
  def run(self):
    dosya=open("%s-%6d-%6d" % (self.name,self.startid,self.finishid),"w")
    for nd_id in range(self.startid,self.finishid):
      bas=time.time()
      Nodes[nd_id].computeBroadcast(Nodes)
      bit=time.time()
      print "%d icin bitirme suresi  %f,   kalan node sayisi %d " % \
      (nd_id, bit - bas,self.finishid - nd_id)
    for nd_id in range(self.startid,self.finishid):
      dosya.write(Nodes[nd_id].printNode())
    dosya.close()

numproc = mpi.size
myid = mpi.rank

print numproc,myid
```

```
numLocalAnts = total_ants / (numproc-1)
cikis = False


mtag = tags()

Nodes =[]
x = 50
y = 50

total_nodes =  x * y

kk = 0
k = False
for i in range(y):
  for j in range(x):
    Nodes.append(Node(kk,i+1.0,j+1.0,(i+1.0)*(j+1.0)))
    kk += 1
Nodes[len(Nodes) - 1].data = TARGET_DATA

length = len(Nodes)
s1 = length / numproc

if myid == numproc - 1:
  bitis = length
else:
  bitis = (myid + 1) *  s1



if (myid == 0):
  req = []
  """
  kutular = []
  xkutuUzunluk = 2 * RADIO_RANGE
  ykutuUzunluk = 2 * RADIO_RANGE

  xkutuSayisi = int(( x / ( xkutuUzunluk )) + 1)
  ykutuSayisi = int(( y / ( ykutuUzunluk )) + 1)

  for i in range(xkutuSayisi):
    kutular.append([])
    for j in range(ykutuSayisi):
      kutular[len(kutular)-1].append([])


  kk=0
  for i in range(y):
    for j in range(x):
      Nodes.append(Node(kk,i+1.0,j+1.0,(i+1.0)*(j+1.0)))
      kk += 1


  for nd in Nodes:
    #print int(nd.x / xkutuUzunluk),int(nd.y / ykutuUzunluk),nd.x,nd.y
    kutular[ int(nd.x / xkutuUzunluk)] [int(nd.y / ykutuUzunluk)].append(nd)
"""

  for dg in range(len(Nodes)):
    Nodes[dg].computeBroadcast(Nodes)
  mpi.barrier()

  for i in range(1,numproc):
    cevap,status = mpi.recv(tag = 1)
    print "m 1>>",status.source, cevap,"tag= ",status.tag

  for i in range(1,numproc):
    cevap,status = mpi.recv( tag = 2)
    print "m 2>>",status.source, cevap,"tag= ",status.tag

  for i in range(1,numproc):
    print "m 3 >> bekliyoruz"
    cevap,status = mpi.recv(tag = 3 )
    print "m 3>>",status.source, cevap,"tag= ",status.tag
```

```
  mpi.barrier()
  print ">> simulation can be started"
  i = 0
  j = 0
  while (cikis == False):
    i += 1
    cvp,stt = mpi.recv()
    if i == (total_ants):
      j += 1
      dosya = open("sonuctxt/step-%05d.txt" % j,"w")
      for n in range(len(Nodes)):
        dosya.write("map %d %d %d %f\n" % \
(j,Nodes[n].x,Nodes[n].y, Nodes[n].ph))
      for n in range(len(Nodes)):
        Nodes[n].evaporate()
      if (j % 10)==0:
        paint(Nodes,j)
        print j
      i = 0
      dosya.close()

    #print ">>> ", stt.source, stt.tag, cvp,stt.source
    if stt.tag == mtag.FINALIZE:
      mpi.finalize()
      sys.exit()
    if stt.tag == mtag.GETMINPH:
      node_id, prevNode = cvp
      minId = Nodes[node_id].minPhNode(prevNode)
      mpi.send(minId,stt.source,tag = mtag.SENDMINPH)

    if stt.tag == mtag.GETMAXPH:
      maxId = Nodes[cvp].maxPhNode()
      mpi.send(maxId,stt.source,tag = mtag.SENDMAXPH)

    if stt.tag == mtag.GETPROBNEXTNODE:
      ant_node , node_id = cvp
      probNextNode = Nodes[node_id].probableNextNode(ant_node)
      mpi.send(probNextNode,stt.source,tag = mtag.SENDPROBNEXTNODE)

    if stt.tag == mtag.GETDATA:
      data = Nodes[cvp].data
      mpi.send(data, stt.source, tag = mtag.SENDDATA)

    if stt.tag == mtag.GETPH:
      mpi.send(Nodes[cvp].ph, stt.source, tag = mtag.SENDPH)

    if stt.tag == mtag.SETPH:
      node_id , newPh = cvp
      Nodes[node_id].setPH(newPh)

    if stt.tag == mtag.UPDATEPH:
      node_id , hopCount = cvp
      Nodes[node_id].updatePh(hopCount)

    if stt.tag == mtag.GETNODE:
      mpi.send([cvp[0],Nodes[cvp[1]]], stt.source, tag = mtag.SENDNODE)


else:
  print "<< %d waiting for initial computation" % myid
  mpi.barrier()
  mpi.send('waiting compute neighborhood',0,tag = 1)

  Ants = []

  print "<< %d waiting for ant creation " % myid
  for x in range(numLocalAnts):
    print "%d   %d numarali karincayi yaratti" % (myid, x)
    Ants.append(Ant(myid * 10000 + x  ,0))

  mpi.send('create ant',0,tag = 2)
```

```
print "<< %d waiting for simulation to start" % myid
# ant create buraya

print len(Ants)
for  antid  in range(len(Ants)):
  print "%d,%d thread start" % (myid,Ants[antid].idx)
  Ants[antid].start()

print "%d start gonder " % myid
mpi.send('start',0,tag = 3)
mpi.barrier()
for  antid  in range(len(Ants)):
  print "%d,%d thread start" % (myid,Ants[antid].idx)
  Ants[antid].startSim = True

print "<< %d simulation started" % myid
```

ÖZGEÇMİŞ

1972 yılında Ankara'da doğdu. İlk ve orta öğrenimini Ankara'da, lise öğrenimini İstanbul'da tamamladı. 1994 yılında girdiği Kocaeli Üniversitesi Bilgisayar Mühendisliği Bölümü'nden 1998 yılında mezun oldu. 2000 yılından beri TUBITAK Ulusal Elektronik ve Kriptoloji Araştırma Enstitüsü'nde Araştırmacı olarak çalışmakta olup, evli ve bir çocuk babasıdır.