FOTONİK AĞLARLA BAĞLI ÇOK ÇEKİRDEKLİ İŞLEMCİLER İÇİN YÜKSEK DÜZEYLİ MİMARİ (HLA) STANDARDINDA VERİ İLETİŞİMİNİN MODELLENMESİ

MODELING OF DATA COMMUNICATION IN HIGH LEVEL ARCHITECTURE-HLA FOR PHOTONIC NETWORK CONNECTED MULTI-CORE PROCESSORS

NEVZAT SEVİM

YRD. DOÇ. DR. KAYHAN M. İMRE

Tez Danışmanı

Hacettepe Üniversitesi

Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliğinin

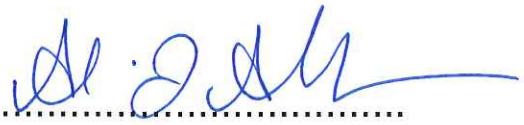
Bilgisayar Mühendisiliği Anabilim Dalı İçin Öngördüğü

YÜKSEK LİSANS TEZİ olarak hazırlanmıştır

2014

NEVZAT SEVİM'in hazırladığı "**Fotonik Ağlarla Bağlı Çok Çekirdekli İşlemciler İçin Yüksek Düzeyli Mimari (HLA) Standardında Veri İletişiminin Modellenmesi**" adlı bu çalışma aşağıdaki juri tarafından **BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Doç. Dr. Ali Ziya ALKAR

Başkan ..... 

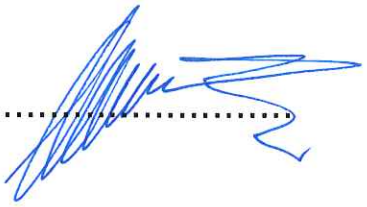
Yrd. Doç. Dr. Kayhan İMRE

Danışman ..... 

Yrd. Doç. Dr. Murat AYDOS

Üye ..... 

Yrd. Doç. Dr. Mustafa EGE

Üye ..... 

Yrd. Doç. Dr. Sevil ŞEN

Üye ..... 

Bu tez Hacettepe Üniversitesi Fen Bilimleri Enstitüsü tarafından **YÜKSEK LİSANS TEZİ** olarak onaylanmıştır.

Prof. Dr. Fatma SEVİN DÜZ

Fen Bilimleri Enstitüsü Müdürü

ETİK

Hacettepe Üniversitesi Fen Bilimleri Enstitüsü, tez yazım kurallarına uygun olarak hazırladığım bu tez çalışmasında,

- tez içindeki bütün bilgi ve belgeleri akademik kurallar çerçevesinde elde ettiğimi,
- görsel, işitsel ve yazılı tüm bilgi ve sonuçları bilimsel ahlak kurallarına uygun olarak sunduğumu,
- başkalarının eserlerinden yararlanılması durumunda ilgili eserlere bilimsel normlara uygun olarak atıfta bulunduğumu,
- atıfta bulunduğum eserlerin tümünü kaynak olarak gösterdiğimi,
- kullanılan verilerde herhangi bir tahrifat yapmadığımı,
- ve bu tezin herhangi bir bölümünü bu üniversitede veya başka bir üniversitede başka bir tez çalışması olarak sunmadığımı

beyan ederim.

01./09./2014

NEVZAT SEVİM

ÖZET

FOTONİK AĞLARLA BAĞLI ÇOK ÇEKİRDEKLİ İŞLEMCİLER İÇİN YÜKSEK DÜZEYLİ MİMARİ (HLA) STANDARDINDA VERİ İLETİŞİMİNİN MODELLENMESİ

Nevzat SEVİM

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Danışmanı: Yrd. Doç. Dr. Kayhan İMRE

Ağustos 2014, 67 sayfa

Çok büyük hesaplamalar gerektiren araştırmalar veya çok fazla işlem yetisi gerektiren benzetimler artık tek çekirdekli işlemcilerle çözülememektedir. İşlemcilerdeki çekirdek sayısı arttırılarak bu tür işlemlerin daha hızlı çözülmesi amaçlanmaktadır. Çok çekirdekli işlemciler kendilerine gelen işlemleri üzerlerindeki çekirdeklere paralel bir şekilde dağıtarak görevlerin daha kısa zamanda bitmesini sağlamaktadır.

Çok çekirdekli işlemciler birçok farklı problemin çözümünde etkin bir şekilde kullanılabilmektedir. Büyük ölçekli benzetimler, çok sayıda işlem gerektiren ileri düzey matematiksel problemler ve hava durumu hesaplama işlemleri gibi birçok problem koşut işlemle çok daha hızlı çözülebilmektedir ve bu problemlerin çözümüne yönelik birçok koşut işlem algoritması geliştirilmiştir. IEEE tarafından dağıtık simülasyonların geliştirilmesi için önerilen HLA (High Level Architecture) standardı ile uyumlu geliştirilen benzetimler de koşut işlem kullanıldığı zaman çok daha hızlı çalışabilmektedir.

High Level Architecture (HLA) bağımsız benzetimlerin entegrasyonunu kolaylaştıran bir standarttır. Bu standart sayesinde farklı platformlarda çalışan benzetimler uyum problemi yaşamadan birbirleriyle haberleşebilmektedir. Benzetimdeki her bir elemana federe denmektedir. HLA üzerinde çalışan RTI mekanizması federeler arasındaki her türlü iletişimi sağlamaktadır. Bu çalışmada RTI'ın federeler arasındaki üye olma mekanizmasını yöneten *deklarasyon yönetimi* ve veri iletişimini yöneten *nesne yönetimi* servisleri için çözüm yaklaşımı anlatılmıştır. Bununla birlikte *nesne yönetimi* servisinin benzetimi yapılarak etkinliği ölçülmüştür. Üstelik önerdiğimiz yöntem, veri dağıtımı için kullanılabilecek etkin yöntemlerden biri olan hasır yöntemi ile karşılaştırılmıştır.

Fotonik ağlar, verilerin çekirdekler arasında elektrik sinyalleri ile değil de ışık sinyalleri ile iletildiği ağlardır. Daha henüz laboratuvar ortamında olsa da, işlemciler üzerindeki çekirdeklerin kendi aralarındaki veri iletişiminde de fotonik ağlar kullanılabilmektedir. Bu ağlar yüksek performansları ve az enerji harcamaları sayesinde geleceğin teknolojisi olarak gösterilmektedir. Bu tez kapsamında önerilen yöntemlerde fotonik ağlar kullanılmıştır.

Bu çalışma ile HLA'de veri iletişimi için önerilen yöntemin olumlu ve olumsuz tarafları incelendi. Yöntemin olumsuzluklarını azaltmak için hasır yapısının nasıl kullanıldığı anlatılmıştır. Sistem gereksinimleri göz önünde bulundurularak hangi yöntemin kullanılmasının daha mantıklı olacağı incelenmiştir.

Anahtar Kelimeler: Koşut İşlem, Yüksek Düzeyli Mimari, Koşum Zamanı Altyapısı, Nesne Yönetimi Servisi, 2B Torus, Fotonik Ağlar, Fotonik Anahtarlar, Veri İletişimi

ABSTRACT

MODELING OF DATA COMMUNICATION IN HIGH LEVEL ARCHITECTURE-HLA FOR PHOTONIC NETWORK CONNECTED MULTI-CORE PROCESSORS

Nevzat SEVİM

Master of Science, Department of Computer Engineering

Supervisor: Assist. Prof. Dr. Kayhan İMRE

August 2014, 67 pages

Simulations that require complicated computational capability or massive calculations cannot be performed without the aid of multi-core processors. Increasing the number of cores in processors is the essential method to overcome such problems. Multi-core processors distribute the incoming processes to each single core and perform parallel computing in order to complete tasks more rapidly.

Multi-core processors can be used effectively in the solution of various problems. Numerous parallel processing algorithms have been developed for the solution of these problems and many problems can be solved much faster by parallel processing; such as large-scale simulations, advanced mathematical problems and weather condition computations. HLA (High Level Architecture) that is recommended for development of distributed simulations by IEEE is a standard which can operate much faster by using parallel processing.

High level architecture (HLA) is a standard that is used for distributed simulations. It facilitates the administration of complicated simulations. By courtesy of HLA, simulations that are running on different platforms can communicate with each other without having compatibility problems. The RTI mechanism which works on HLA provides all kinds of communications among federates. Any application in the simulation is called a federate. In this study, methods are proposed for *declaration service* which manage the RTI's subscription mechanism among federates and *object management* service which leads the data communication. In addition, object management service is implemented and tested. Furthermore, the method that we suggest is even compared with mesh method which is considered to be one of the effective methods for data distribution.

Photonic networks is the networks where the data is transmitted in between cores by light signals instead of electrical ones Even in the laboratory yet, photonic networks can be used for data communication in between cores seated on the chip. These nework is shown as the future technology due to high performance and low energy consumption. The routing problem of data between cores can be solved efficiently with photonic networks. In this thesis scope, photonic networks has been used in the proposed methods.

In this study, we have examined the advantages ans disadvantages of the pattern that we recommend for data communication in RTI. To reduce the disadvantages of the pattern, we explain how we use mesh method. By considering the system requirements, we have evaluated which method is more reasonable.

Keywords: Parallel Processing, HLA-High Level Architecture, RTI- Run-Time Infrastructure, Object Managment Service, Photonic Networks, Photonic Switches, 2D Torus, Data Communication

TEŞEKKÜR

Öncelikle, beni üniversite son sınıftayken akademik hayata ısındıran, yüksek lisans eğitimim sırasında sürekli destekleyen, tez çalışmaları sırasında her türlü yardımı ve bilgiyi sağlayan, beni sürekli anlayışla karşılayan ve o samimi yaklaşımını hiç esirgemeyen danışman hocam Sayın Yrd. Doç. Dr. Kayhan İMRE'ye özel olarak teşekkür ederim.

Yüksek lisans eğitimim sırasında beraber olduğumuz, beni yalnız bırakmayan, sürekli destekleyen ve cesaretlendiren Abdülkadir YAŞAR'a ve Ahmet AYŞAN'a teşekkür ederim.

Ayrıca, yüksek lisans eğitimim sırasında beni sürekli destekledikleri ve maddi-manevi her türlü yardımı yaptıkları için özellikle aileme teşekkür ederim.

Son olarak lisans ve yüksek lisans eğitimlerim sırasında beni maddi olarak destekleyen ve akademik hayata teşvik eden, şu anda bir çalışanı olarak gurur duyduğum, TÜBİTAK'a teşekkür ederim.

İÇİNDEKİLER

Sayfa

SİMGELER VE KISALTMALAR

Simgeler

α	Alfa-Paket Hazırlama Süresi
l	Mesaj Boyu
τ	Tau- Bir Bayt Veriyi İletim Hızı
E_p	Verimlilik
S_p	Hızlanma Değeri
P	İşlemci Sayısı
sn	Saniye
ms	Milisaniye

Kısaltmalar

Fib	Fibonacci Sayıları
HLA	High Level Architecture - Yüksek Düzeyli Mimari
SISD	Single Instruction, Single Data
MISD	Multiple Instructions, Single Data
SIMD	Single Instruction, Multiple Data
MIMD	Multiple Instruction, Multiple Data
RTI	Run-Time Infrastructure - Koşum Zamanı Altyapısı
UAV	Update Attribute Value
S OCA	Subscribe Object Class Attribute - Nesne Sınıfı Özelliğine Üye Ol
GALT	Greatest Available Logical Time - Uygun En Büyük Mantıksal Zaman

1. GİRİŞ

Günümüzde, tek çekirdekli işlemciler yerlerini dört, sekiz, on altı hatta çok daha fazla çekirdekli işlemcilere bırakmıştır [8,9]. Halihazırda üretilen ve kullanılan en fazla 60 çekirdekli işlemciler vardır [26]. Gerek çok fazla işlem yetisi gerektiren benzetimlerde gerekse de büyük hesaplama gerektiren bilimsel araştırmalarda bu çok çekirdekli işlemciler yoğun bir şekilde kullanılmaktadır [17]. Bilgisayarların daha hızlı çalışmasını sağlamak için çalışan üreticiler bu bilgisayarların işlemcilerine yerleştirilen çekirdek sayılarını artırmaya çalışmaktadırlar.

Çok çekirdekli işlemcilere atanan işlemler, işlemci üzerindeki çekirdeklere paralel bir şekilde dağıtılarak, görevlerin daha kısa zamanda bitmesi sağlanmaktadır. Burada karşılaşılan en büyük problemlerden biri, bu işlemleri uygun bir şekilde alt görevlere bölmek, bu alt görevleri çekirdeklerde paralel olarak çalıştırmak ve çalıştırılan görevlerin sonuçları birleştirilerek asıl büyük problemin çözümlenmesidir. Bu sürece basit bir örnek vermek gerekirse Fibonacci Sayıları incelenebilir. Fibonacci Sayıları hesaplanırken Formül 1.1 kullanılır. Bu formülde x. Fibonacci Sayısını bize veren $fib(x)$ fonksiyonu, $fib(x-1)$ ve $fib(x-2)$ şeklinde iki alt probleme ayrılmıştır. Bu alt problemler birbirlerinden bağımsız oldukları için farklı çekirdeklerde paralel olarak işletilirler. Bu alt problemler çözümlendikten sonra toplanarak birleştirilir ve asıl bulmak istediğimiz $fib(x)$ bulunmuş olur. Bu sayede Fibonacci Sayıları çok daha hızlı hesaplanabilmektedir.

$$fib(x) = fib(x-1) + fib(x-2)$$

Formül 1.1 Fibonacci Sayılarının hesaplanması

Bir başka karşılaşılan problem ise; işlemler paralel bir şekilde çekirdekler üzerinde işlenirken, çekirdeklerin kendi aralarında veri alışverişinde bulunma gereksinimleridir. Bu gereksinimden dolayı işlemciler başka bir işlemciden veri beklerken boşta bekler. Bu

kaynakların etkin kullanılmamasına ve işlemlerin istenilen hızda çözülememesine sebep olmaktadır.

Bu problemler göz önünde bulundurulduğunda, çekirdekler arasındaki veri aktarım yollarının tasarlanması ve yönetilmesi çok çekirdekli işlemciler için önem arz etmektedir. İyi bir şekilde tasarlanıp yönetilen trafikler işlemci üzerindeki iletişim kaynaklı gecikmeleri azaltarak, işlemcilerin daha verimli kullanılmasını sağlayacaktır.

Çok çekirdekli işlemciler birçok farklı problemin çözümünde etkin bir şekilde kullanılabilmektedir. Büyük ölçekli benzetimler, çok işlem gerektiren ileri düzey matematiksel problemler ve hava durumu hesaplamaları gibi birçok problem koşut işlemle çok daha hızlı çözülebilmektedir. Bu problemlerin çözümüne yönelik birçok koşut işlem algoritması geliştirilmiştir.

IEEE tarafından en son 2010 yılında güncellenen, birbirinden bağımsız dağıtık benzetimlerin entegrasyonu için kullanılan Yüksek Düzeyli Mimari (High Level Architecture – HLA) bir standarttır. Bu standart ile uyumlu gerçekleştirilen benzetimler koşut işlem ile birlikte çok daha hızlı çalışabilmektedir [10,11,12]. HLA'de benzetim sırasında, federeler arasındaki veri aktarımı çok önemlidir. Kaynakların (veri yolları, işlemciler vb.) etkin kullanılamaması, boşta beklemelerin sistemi yavaşlatması gibi problemler veri dağıtımı sırasında ortaya çıkabilecek problemlerdir. Bu tür problemlerden dolayı HLA'de veri dağıtımı ne kadar etkin çözülürse sistem o kadar hızlı çalışacaktır. Bu çalışma kapsamında çok çekirdeki işlemci mimarileri için HLA'de veri aktarımını etkin bir şekilde gerçekleştiren yöntem önerilmiş ve bu yöntemin kodu yazılarak gerçekleştirimi yapılmıştır.

Geleceğin teknolojisi olarak gösterilen, işlemciler üzerindeki çekirdeklerin fotonik ağlar ile iletişim kurmaları şu anda laboratuvar ortamında gerçekleştirilmiş ve üzerindeki çalışmalar devam etmektedir

[6]. Bu ağlar enerji tüketimini azaltan ve daha hızlı veri aktarımını sağlayan ağlardır. Çalışmamız bu tür işlemciler üzerinde yürütülmüştür.

[7] nolu çalışmadaki iletişim örüntüleri, işlemciler arasındaki her yerden-her yere (all-to-all) veri aktarımını etkin bir şekilde gerçekleştirebilen bir yöntemdir. Bu çalışmada, HLA'de veri dağıtımının modellenmesi, iletişim örüntüleri ile 2 boyutlu fotonik toruslar kullanılarak gerçekleştirilmiştir. Bu gerçekleştirim sonucunda elde ettiğimiz verilerle örüntülerin etkinlikleri ölçülüp analizleri yapılmıştır.

Bu tezin 2'nci bölümünde koşut işlem ile alakalı genel bilgiler, 3'üncü bölümünde HLA (High Level Architecture) standardı hakkında bilgiler, 4'üncü kısmında fotonik ağlar, bu ağlar üzerinde veri yönlendirilmesi için kullanılan fotonik anahtarlar (photonic switches) ve hatların etkin bir şekilde kullanıldığı bir fotonik anahtar tasarımından bahsedilecektir. 5'inci bölümde HLA'de veri dağıtımını gerçekleştirmede kullanılacak yöntemlerin açıklaması, 6'ncı bölümde ise bu yaklaşımların benzetimi ve bu benzetimin neticesinde elde ettiğimiz sonuçları bulabilirisiniz.

2. KOŞUT İŞLEM NEDİR?

Günümüzün modern işlemcileri saniyede milyarlarca işlem yapabilmektedir. Buna rağmen birçok problemin çözümü böyle işlemcilerle bile çok uzun zaman sürmektedir. Benzetimler, çok işlem gerektiren ileri düzey matematiksel problemler, hava durumu hesaplama işlemleri ve çok işlem gerektiren bilimsel çalışmalar bu tür işlemlerden bazılarıdır.

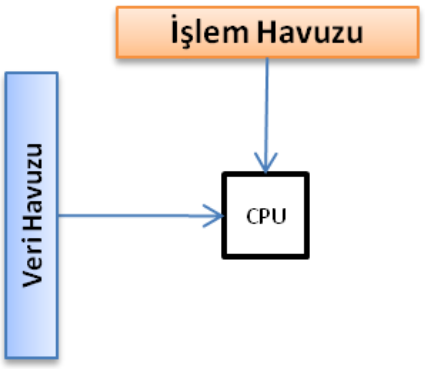
Birçok bilim dalı bu tür problemleri nasıl daha hızlı çözebiliriz diye çalışmaktadır. Üzerinde çalışılan yöntemlerden biri de aynı anda birden fazla işlem birimi kullanarak bu problemleri, paralel olarak bu birimler üzerinde çözmektir. Bu işleme koşut işlem denmektedir.

Koşut işlem şöyle de tanımlanabilir: En az iki işlemci kullanarak, bütüncül problemi alt problemler olarak çözmektir. Bunun için karmaşık problem, daha küçük alt problemlere ayrılır sonra her bir alt problem özel yazılım ve donanımlar kullanılarak farklı işlemcilere çözdürülürler. Her bir işlemci kendi üzerine düşen görevi tamamladıktan sonra, bu alt çözümler bir araya getirilerek asıl büyük problemin çözümü gerçekleştirilir.

Bu bölümde koşut işlemin daha iyi anlaşılması için bilgisayar bilimcileri tarafından tanımlanan dört temel işlemci mimarisi açıklanacaktır. Bu mimariler en temelde instruction streams (işletilen komut) ve data streams (işlenen veri) sayılarına göre şekillenmiştir. Instruction streams veriler üzerinde işlem yapan komutlar, data streams ise işlemcide işlenen veriler olarak algılanabilir [3].

Tekil Komut, Tekil Veri (Single Instruction, Single Data-SISD)

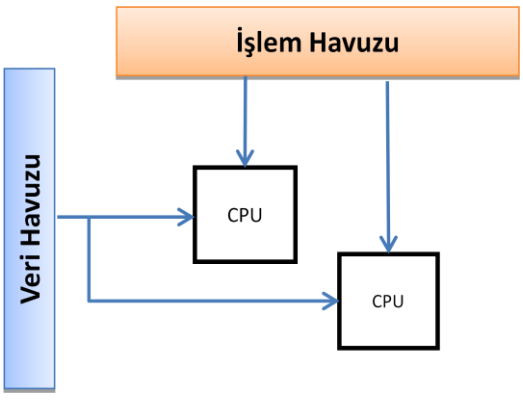
Bu mimaride, işlemci üzerinde aynı anda bir işlem yapılabilmektedir ve işlemci aynı anda sadece bir veri üzerinde çalışabilmektedir. Bu mimaride yapılacak işler sırayla işlemci üzerinde çalıştırılır.



Şekil 2.1 SISD Mimarisi

Çoklu Komut, Tekil Veri (Multiple Instructions, Single Data-MISD)

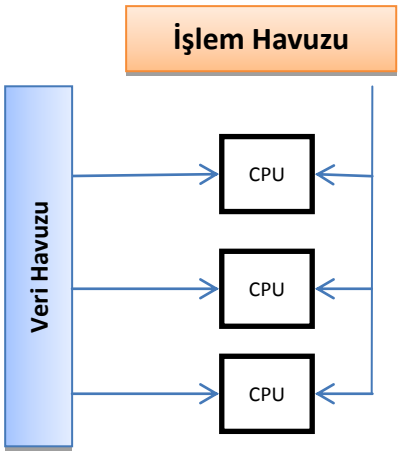
Bu mimaride birden fazla işlemci yer almaktadır ve aynı anda birden farklı işlem işletilebilmektedir. Ama aynı anda tek veri üzerinde işlem yapılabilmektedir. Bu sayede aynı anda birden fazla işlemci tek veri üzerinde işlem yapabilmektedir. MISD mimarisinde aynı veri aynı anda farklı analizlere tabi tutulabilir, bu analizlerin sayısı sistemdeki işlemci sayısına bağlıdır.



Şekil 2.2 MISD Mimarisi

Tekil Komut, Çoklu Veri (Single Instruction, Multiple Data-SIMD)

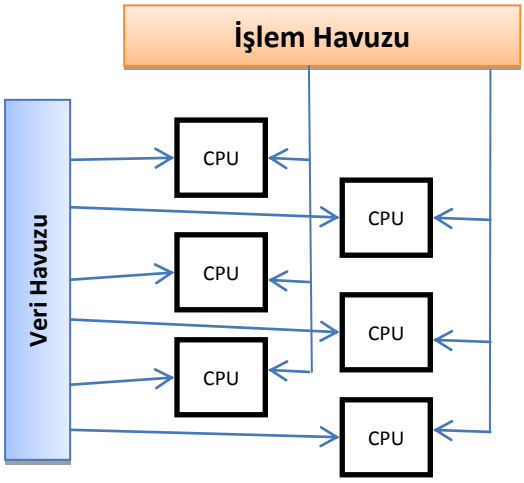
Bu mimaride bilgisayarın birden fazla işlemcisi olmasına rağmen, işlemciler aynı tür işlemleri yapmaktadır. Farklı işlemcilerde farklı veriler işlenebildiği için aynı anda farklı veriler işleme tabi tutulabilmektedir.



Şekil 2.3 SIMD Mimarisi

Çoklu Komut, Çoklu Veri (Multiple Instruction, Multiple Data-MIMD)

Bu mimaride bilgisayarlar birden fazla işlemciye sahiptir ve işlemciler birbirlerinden farklı işlem yetilerine sahiptirler. Bununla birlikte her işlemci farklı verileri işleyebilir. Bu sayede aynı anda birden fazla işlemcide, birden fazla veri işlenebilir.



Şekil 2.4 MIMD Mimarisi

Yukarıda belirtilen mimarilerden ilki hariç diğer üçü koşut işlem yapmaya müsait yapılardır. Özellikle MIMD ve SIMD mimarileri koşut işlem için en çok kullanılan mimarilerdir. SISD mimarisi ise tek başına koşut işleme müsait bir mimari olmasa da birden fazla böyle işlemci ağ yapısı ile birbirine bağlanarak oluşturulan sistemler, kullanılacak yazılımlar ile koşut işlem yapılabilir bir hal alabilmektedir [1].

Koşut işlem yapılırken kaynakların verimli kullanılması çok önemlidir. Seri yapılan işlemlerde işlemciler arasında veri alışverişi olmadığı için işlemci beklemek zorunda kalmaz ve %100 verimle çalışabilir. Ama koşut işlem sırasında, işlemciler arasındaki veri alışverişleri işlemcilerin boşta beklemesine sebebiyet vermektedir. Bu boşta beklemeler ne kadar kısa olursa sistem o kadar verimli çalışıyor demektir.

Bir problemin çözümünde, koşut işlemler için tasarlanmış bir yapının, seri yapıya göre kaç kat daha hızlı çalıştığına ***hızlanma değeri*** denmektedir. Bu değerin, koşut işlem için kullanılan işlemci sayısına oranına ise ***verimlilik*** denmektedir [22].

Sistemin verimliliği aşağıdaki formülle hesaplanabilir.

$$E_p = \frac{S_p}{P}$$

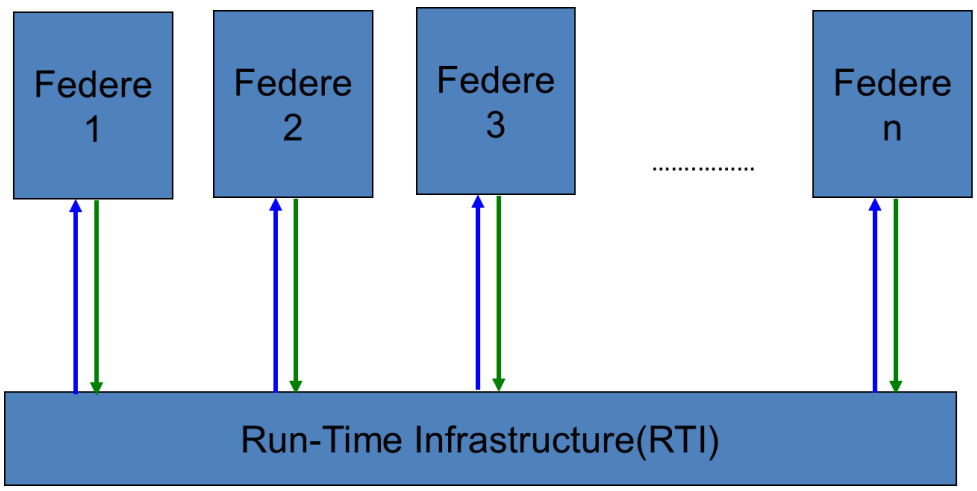
$$S_p = \frac{\text{En iyi seri algoritmanın çalışma zamanı}}{\text{Koşut algoritmanın çalışma zamanı}}$$

E_p -> Verimlilik S_p -> Hızlanma değeri P -> İşlemci sayısı

Formül 2.1 Hızlanma Değeri ve Verimlilik hesaplanması

3. YÜKSEK DÜZEYLİ MİMARİ (HIGH LEVEL ARCHITECTURE-HLA)

High Level Architecture (HLA) dağıtık benzetimler için kullanılan, birbirinden bağımsız benzetimlerin birlikte yönetimini sağlayan bir standarttır [13]. Bu standart sayesinde farklı platformlarda çalışan benzetimler uyum problemi yaşamadan birbirleriyle haberleşebilmektedir. HLA standardını iki önemli kavram üzerinden inceleyelim: Bunlardan ilki ***federe*** kavramıdır. Federe, benzetimdeki her bir öğeyi temsil etmektedir. Bu öğeler, benzetimdeki nesneleri yönetmektedir. Örneğin; bir savaş benzetimi yapıyorsanız tanklar, askerler, mermiler gibi benzetimdeki tüm elemanlar nesnedir. Bu tür nesnelerin birini veya birkaçını yöneten uygulamalara ise federe denir. İkinci önemli kavram ise, **Koşum Zamanı Altyapısı (Run-Time Infrastructure-RTI)** kavramıdır. RTI, federeler arasındaki tüm veri iletişimini sağlayan yapıdır. Federeler arasındaki veri aktarımı, zaman uyumu, hangi federelerin birbirleriyle iletişim içinde olacağı gibi tüm veri yönetimi RTI sayesinde sağlanmaktadır [13]. Bu bölümün ilerleyen kısımlarında RTI hakkında daha ayrıntılı bilgiler verilecektir.



Şekil 3.1 Federelerin RTI üzerinden iletişime geçmesi

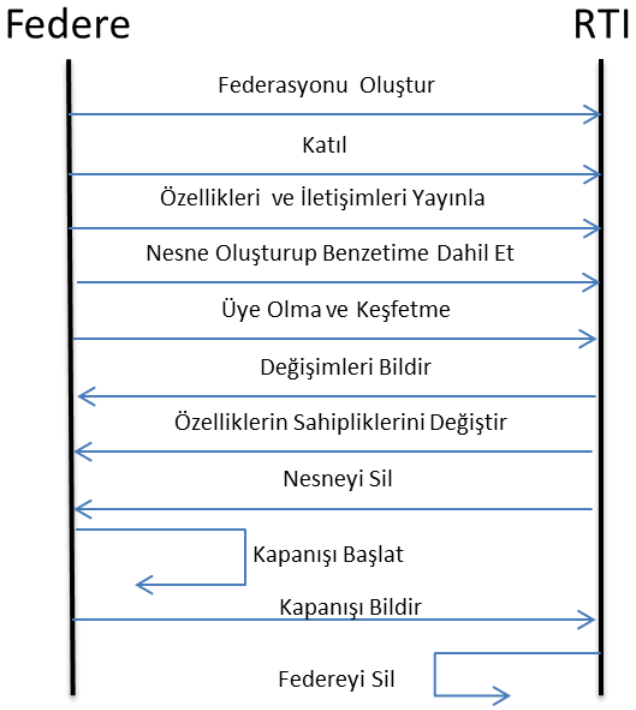
IEEE tarafından belirlenen bu standart en son 2010 yılında güncellenmiştir. Birçok alana uygulanabilen bu standartın kullanıldığı alanlardan bir kısmı şöyle sıralanabilir:

- Savunma Sanayii
- Hava Trafiği Yönetimi
- Uzay Bilimi Araştırmaları
- Sağlık
- Enerji Sanayii
- İmalat Sanayii
- İlaç Sanayii
- Oyunlar

HLA temelde 3 bölümden oluşmaktadır;

1. **Ara Yüz Tanımlamaları:** Bu bileşen sayesinde benzetim elemanları RTI ile iletişime geçmektedir.
2. **Nesne Modeli Şablonu (OMT):** Benzetim sırasında kullanılacak nesnelerin tanımlamalarıdır.
3. **Kurallar:** Simülasyon sırasında federelerin uyması gereken kurallar.

HLA'de federelerin arasındaki her türlü iletişimi RTI sağlamaktadır. Şekil 3.2 de bir federenin hayat döngüsünü ve bu süreçte RTI ile olan iletişimleri gösterilmiştir [18].



Şekil 3.2 Federe hayat döngüsü ve RTI ile iletişimleri

RTI, federeleri ve federeler arasındaki ilişkileri yönetirken aşağıdaki altı servisi kullanmaktadır [16].

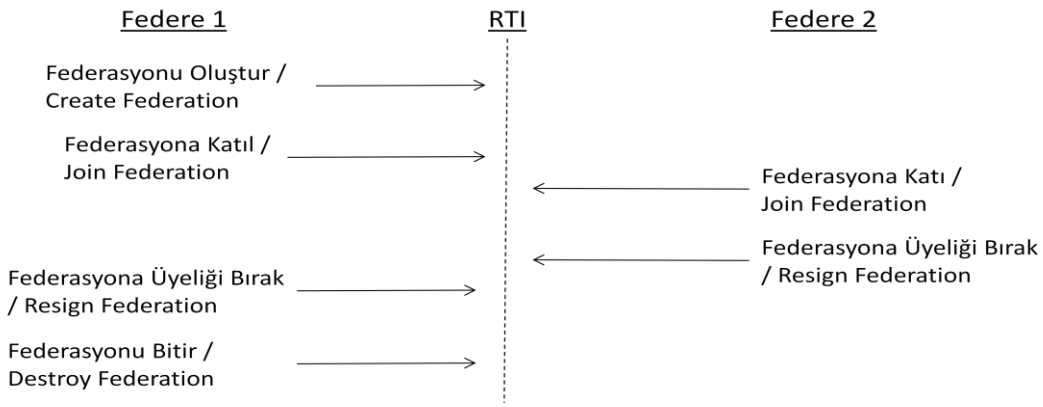
- Federasyon Yönetimi - Federation Management (FM)
- Deklarasyon Yönetimi - Declaration Management (DM)
- Nesne Yönetimi - Object Management (OM)
- Sahiplik Yönetimi - Ownership Management (OwnM)
- Zaman Yönetimi - Time Management (TM)
- Veri Dağıtımı Yönetimi - Data Distribution Management (DDM)

Servisler açıklanmadan önce iki önemli kavramdan söz edilecektir. Bunlar, üye olma (subscribe) ve yayınlama (publish) kavramlarıdır. Benzetimdeki federeler kendi görevlerini yaparken başka federelerin özelliklerine ihtiyaç duyarlar. Örneğin, bir savaş benzetiminde radar nesnelerini yöneten federe uçak, tank gibi diğer nesnelerin konum bilgilerine ihtiyaç duyar. Yönettikleri niteliklerin başka federeler tarafından erişilebilir olmasını isteyen federeler ilgili niteliklerini *yayınlarlar*. Başka federelerin yönettikleri niteliklere ihtiyaç duyan

federeler ise ilgili niteliğe *üye olurlar*. HLA'de *üye olma* ve *yayınlama* işlemleri 5'inci bölümde ayrıntılandırılacaktır.

3.1. Federasyon Yönetimi Servisi

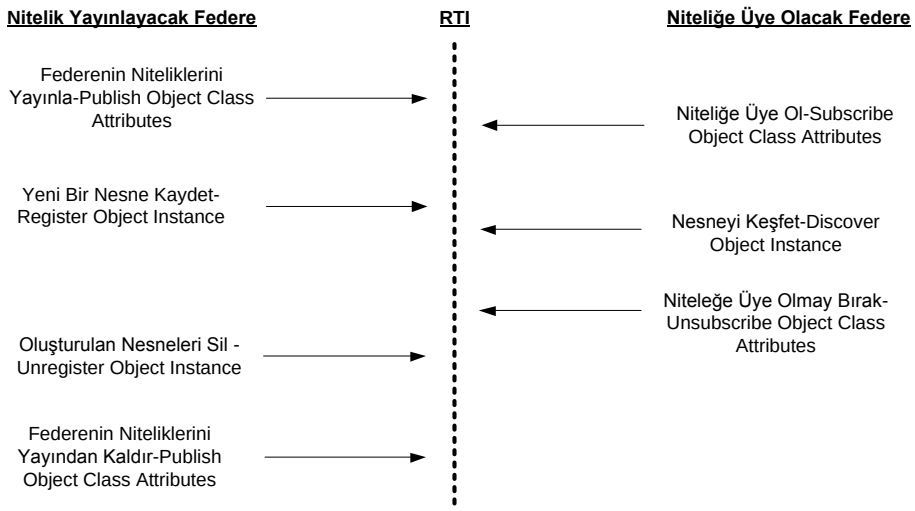
Bu servis federe ile RTI arasındaki iletişimin kurulmasını ve bitirilmesini yönetmektedir. Bu servisle birlikte federe ile RTI arasında bir federasyon kurulur. Yine federe, RTI'ya federasyonu bitirmek istediğini bu servis sayesinde bildirir. Şekil 3.3'de bu servis için federelerin RTI ile yaptıkları iletişimler görülmektedir [23].



Şekil 3.3 Federasyon yönetimi servisi hayat döngüsü

3.2. Deklarasyon Yönetimi Servisi

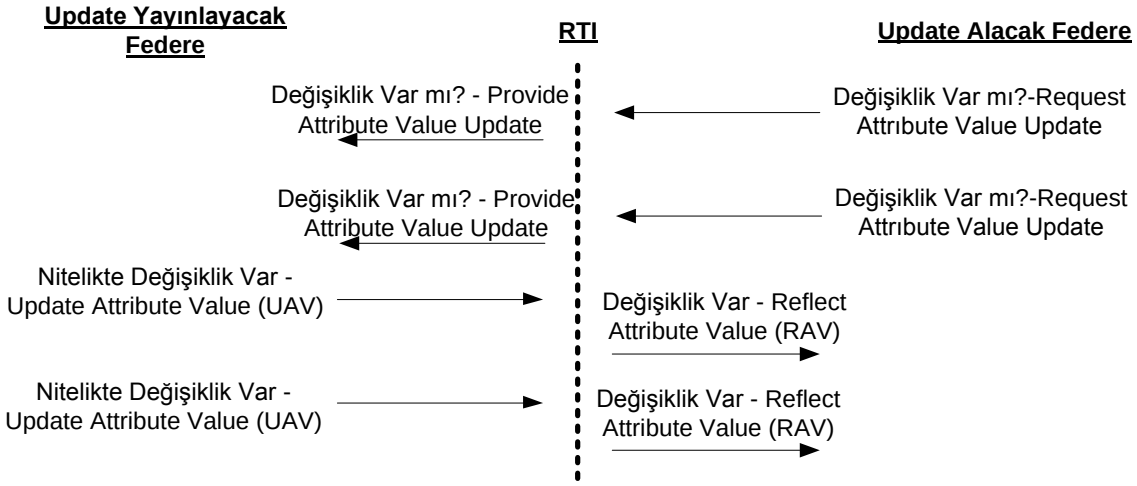
Bu servis, federelerin yönettiği nesnelerin üzerlerindeki niteliklerini (attribute) yayınlamalarını ve başka nesnelerin niteliklerine üye olmalarını sağlar. Tez kapsamında bu servis için bir çözüm yaklaşımı önerilmiştir. Bu servis 5'inci bölümde ayrıntılandırılacaktır. Şekil 3.4'de deklarasyon yönetimi servisinin hayat döngüsü gösterilmiştir [23].



Şekil 3.4 Deklarasyon yönetimi servisi hayat döngüsü

3.3. Nesne Yönetimi Servisi

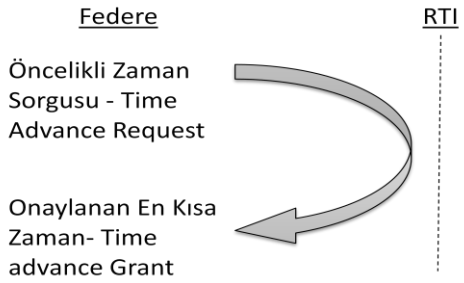
Nesne yönetimi servisi, nesnelerin yayınladıkları niteliklerindeki değişikliklerin duyurulmasını ve federelerin üye oldukları niteliklerin değişimlerini almalarını sağlar. Tez kapsamında bu servis için çözüm yaklaşımı anlatılmış ve bu servisin benzetimi gerçekleştirilmiştir. 5'inci bölümde bu servis ayrıntılandırılmıştır. Şekil 3.52'de bu servis için federelerin RTI ile olan iletişimleri verilmiştir [23].



Şekil 3.5 Nesne yönetimi servisi hayat döngüsü

3.4. Zaman Yönetimi Servisi

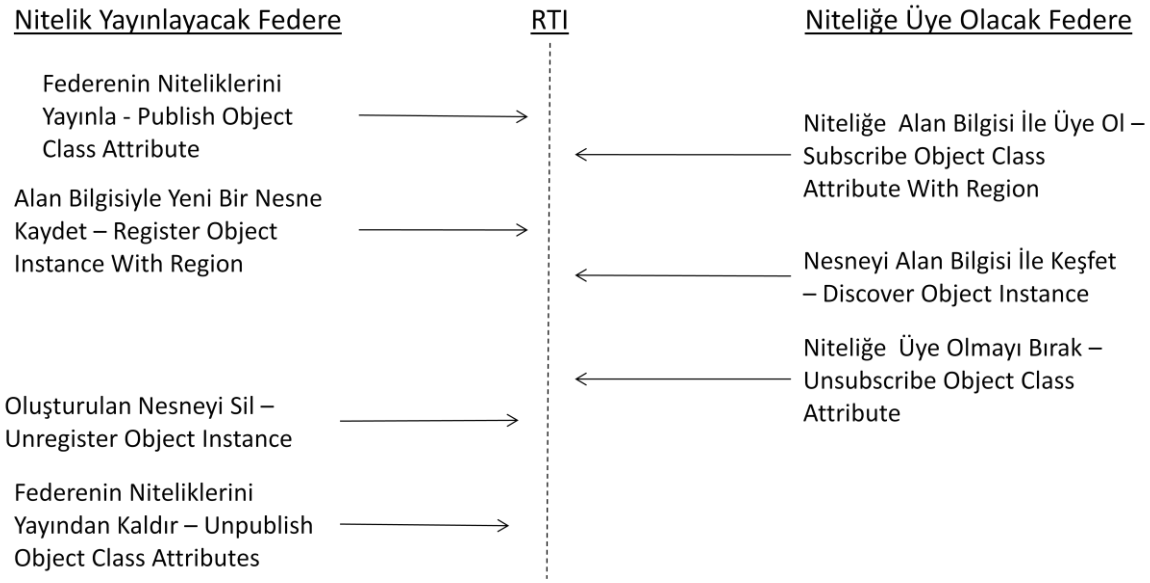
Federelerin zaman uyumunu sağlayan servistir. Bu servis ayrıca mantıksal zamanın ne olması gerektiğini belirler. Şekil 3.6'da bir federenin RTI'dan zaman sorgusu yapması ve bu sorgu sonucu aldığı yanıt verilmiştir [23].



Şekil 3.6 Zaman yönetimi servisi federe-RTI ilişkisi

3.5. Veri Dağıtım Yönetimi Servisi

Bu servis, üye olma ve yayınlama işlemleri için RTI'a esnek bir yapı sağlamaktadır. Bu servis sayesinde benzetim bölgelere ayrılarak veri dağıtımı çok daha etkin yapılabilmektedir. Şekil 3.7'de veri dağıtım servisi için hayat döngüsü verilmiştir [23].



Şekil 3.7 Veri dağıtım yönetimi servisi hayat döngüsü

3.6. Sahiplik Yönetimi Servisi

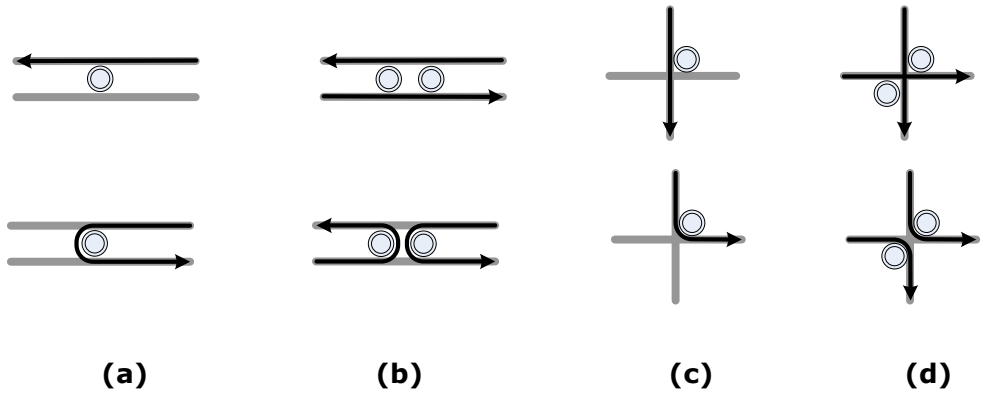
Benzetim sırasında bazı kısıtlamalar veya zorunluluklar neticesinde federeler kendi üzerlerindeki nitelikleri, nesneleri başka federelerin sorumluluğuna vermeleri gerekebilir. Bunun gibi sorumluluğu başka federelerden alabilirler. Sahiplik yönetimi, bu sorumulukları ve değişimlerini yönetmektedir.

4. FOTONİK AĞLAR VE ANAHTARLAR

Fotonik ağlar, verilerin iletişim yolları arasında elektrik sinyalleri ile değil de ışık sinyalleri ile iletiminin sağlandığı ağlardır. Bilgisayarlar arasında fotonik ağların kullanılması yaygın olsa da işlemcilerin kendi aralarındaki ve işlemci üzerindeki çekirdekler arasındaki iletişimler henüz laboratuvar ortamında gerçekleştirilmiştir. Bu tür ağlar gelecekte geniş bir kullanım alanına sahip olacaktır [6]. Fotonik ağların bant genişliği yüksek ve enerji tüketiminin, çok düşük olması kullanımları için başlıca nedenlerdendir.

Bu tür ağlarda, veriler iletişim yollarından iletilirken fotonik ağlar kullanılmasına rağmen verilerin yönlendirilmesinin elektronik yönlendiricilerle yapılmsı sıkça karşılaşılan bir yöntemdir. Elektronik yönlendiriciler araya girdiği zaman fotonik ağ, elektronik yönlendiriciye bağlı hale gelmektedir. Bunun sonucunda fotonik ağların hızlı iletişim, yüksek bant genişliği gibi birçok faydası kaybolmaktadır. Bu çalışmada elektronik yönlendiriciler yerine, fotonik paketleri yönlendirebileceğimiz mikro halkalar kullanılacaktır [4,5].

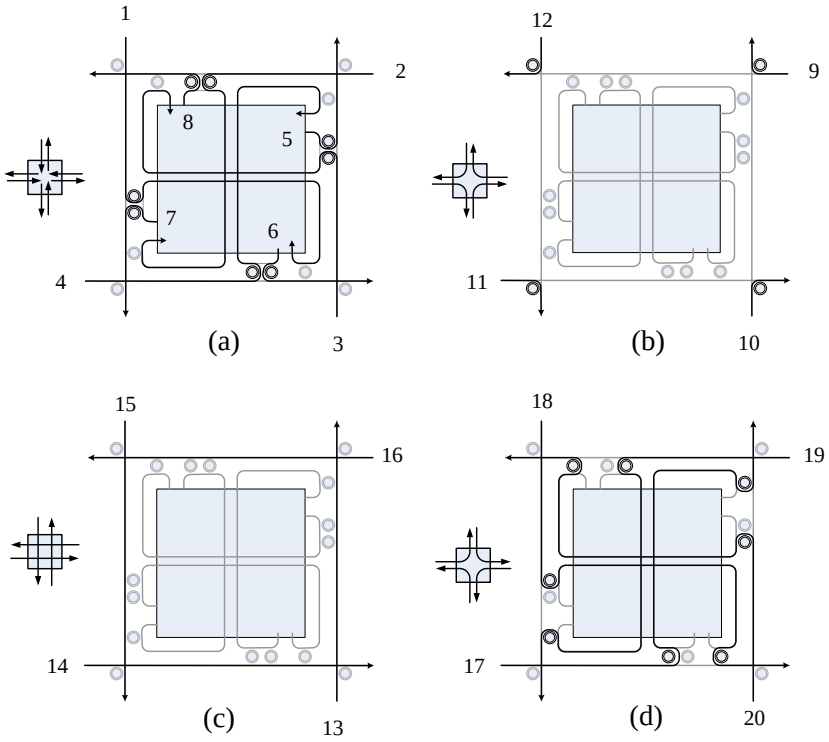
Mikro halkalar, fotonik ağlarda kablolardan geçen verinin yönlendirilmesi için kullanılmaktadır [4,5]. Bu halkalar manyetik özelliklerinden dolayı aktif durumda oldukları zaman fotonları bir kablodan diğer kabloya aktarabilmektedir. Şekil 4.1'de mikro halkaların aktif (alttaki şekiller) ve pasif (üstteki şekiller) durumları için fotonların yönlendirilmesi gösterilmiştir. Bu şekilde 1x2 ve 2x2 yönlendirme yapan basit anahtar yapıları görülmektedir [4,5].



Şekil 4.1 Manyetik halkalar kullanılarak, (a) şeklinde 1x2, (b) şeklinde 2x2, (c) şeklinde 1x2 dikey ve (d) şeklinde 2x2 dikey yönlendirme örneklendirilmiştir.

En basit haliyle şekil 4.1'de verilen manyetik mikro halkaların çeşitli birleşimleri ile veriler her türlü yöne yönlendirilebilmektedir. Şekil 4.2'deki yapı ile birlikte 8x8 bir anahtar gerçekleştirilmiştir [7]. Bu anahtar yapısında sadece 16 tane mikro halka kullanılarak, 4 yönden anahtara gelen veriler farklı kombinasyonlarda halkaların aktif edilmesiyle birlikte herhangi 4 yöne yönlendirilerek anahtardan çıkabilmektedir. Bununla birlikte, herhangi bir yönden gelen veri işlemciye alınabilmekte, üstelik aynı anda tüm yönlerden veri gelse bile bu verilerin hepsi işlemciye aynı anda alınabilmektedir. İşlemciden çıkan veriler için de aynı şeyi söyleyebiliriz, aynı anda işlemci 4 tarafındaki işlemcilerin hepsine veri gönderebilir.

Şekil 4.2-a,b,c,d durumlarında koyu renkli mikro halkalar aktif durumdadır. Bu durumlarda, verilerin aynı anda hangi yönlere yönlendirildiği işlemcilerin yanındaki küçük şekillerde verilmiştir. Örneğin 4.2-a şeklinde 1 numaralı yönden (üstten) giren veri işlemcinin içine girmektedir. Şekil 4.2-b'de 9 numaralı hattan gelen veri (sağdan) ilk mikro halkadan etkilenerek yukarı doğru yönlenmektedir. 4.2-c şeklinde hiçbir halka aktif durumda olmadığı için gelen veriler doğrudan karşı taraftan çıkmaktadır. Bunlar gibi gelen veriler mikro halkaların aktiflik-pasiflik durumlarına göre istenilen yöne yönlendirilebilirler.



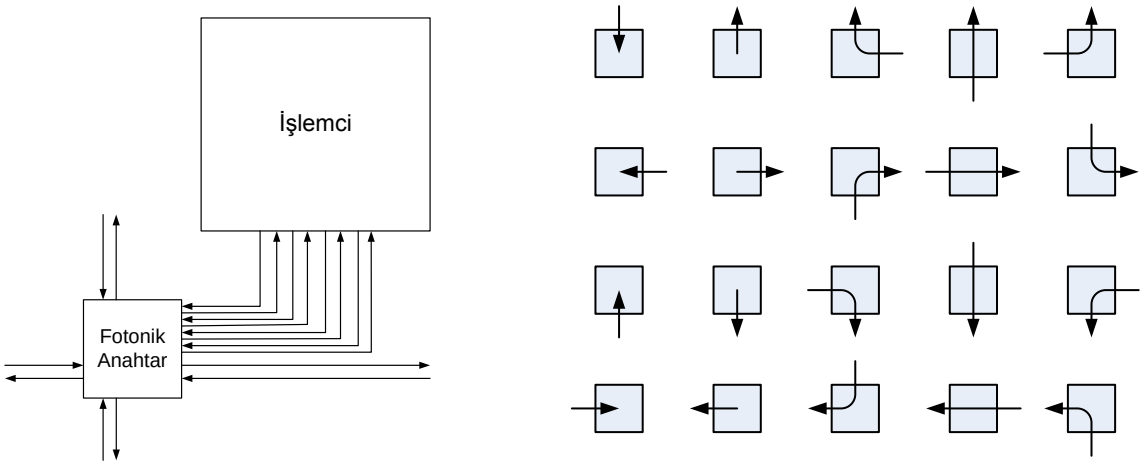
Şekil 4.2 Mikro halkalar kullanarak her yönde gelen ışığı herhangi bir yöne yönlendirebilen anahtar yapısı

Fotonik ağlarda çözüm bekleyen bir diğer konu, paketlerin rotalama problemidir. Veri paketinin bir işlemciden başka bir işlemciye hangi yolu kullanarak iletileceğini belirlemek işlemcilere ek masraflar getirmektedir. Bunun için farklı yaklaşımlar geliştirilmiştir. Bunlardan bir tanesi: veri paketi gönderilmeden önce, farklı bir katmanda elektronik ağlarla rotalamanın yapılmasıdır [24]. Bu tür ağlarda iki ayrı ağ katmanı olur. İlk katmanda elektronik ağ kullanılır ve bu katmanda paketlerin iletileceği rotalar belirlenir. Rotalar belirlenirken mümkün olduğunca küçük paketler kullanılarak iletişim masrafı azaltılmaya çalışılır. Rotalar belirlendikten sonra ikinci katman olan fotonik ağda, mikro halkalar bu rotalara göre kurulur. Son olarak ilgili veri paketi gönderilerek istenilen işlemciye iletilir.

Bu tez kapsamında rotalama probleminin çözümü için farklı bir yaklaşım önerilmektedir. Önceden hazırlanmış basit rotalama seçenekleri iletişim örüntüsü olarak tanımlandıktan sonra bu iletişim örüntüleriyle çakışmasız (contention free) ve kilitlenmesiz (deadlock free) iletişim

sağlanabilmektedir [7]. İleri ki bölümlerde detaylandırılacak olan bu yaklaşım sayesinde tüm iletişim adımlara ayrılmaktadır. Her bir adımda veri yolları iletişim örüntülerine göre belirlenmektedir. Bu sayede rotalama işlemi için işlemcilerin fazladan masraf harcamasına gerek kalmaz. Bu örüntüler sayesinde birkaç adımda tüm işlemcilerden tüm işlemcilere veri aktarılabilmektedir.

Tez kapsamında önerilen çok çekirdekli işlemcilerin ağ mimarisini oluşturmada kullanılan yapı blokları şekil 4.3'deki gibi düşünülmektedir. Bu tasarım ile birlikte her işlemci aynı anda dört farklı işlemciyle iki yönlü iletişim kurabilmektedir.



(a) (b)

Şekil 4.3 İşlemci-Anahtar yapılanması ve bu yapıyla verilerin işlemciye gelebilecek ve işlemciden gönderilebilecek yönleri.

5. FOTONİK AĞ İLE BAĞLI ÇOK ÇEKİRDEKLİ İŞLEMCİLER İÇİN YÜKSEK DÜZEYLİ MİMARİ'DE İLETİŞİM ALTYAPISININ MODELLENMESİ

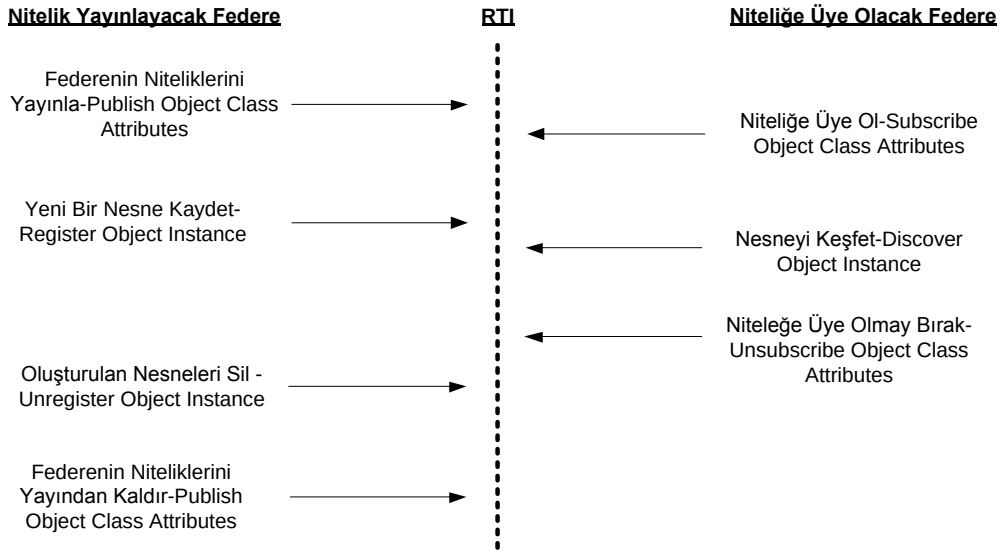
Bu bölümde HLA'de veri iletişimi için nasıl bir altyapı öngürüldüğü anlatılacaktır. HLA'de birden çok veri türü bulunmaktadır. RTI'ın yönettiği servisler farklı türdeki veri türlerinin iletimini ve yönetimini sağlamaktadır. Örneğin, zaman yönetimi servisi benzetimdeki zamanla alakalı verileri yönetir. Uygun En Büyük Mantıksal Zaman (Greatest Available Logical Time-GALT) hesaplaması, zaman sorguları ve bunlara verilen cevaplar gibi iletişimler zaman yönetimi servisi tarafından yönetilir [13,18].

HLA ile uyumlu benzetimlerde en çok görülen iletişim türleri; üye olma (subscribe), yayınlama (publish) ve nesnelerin niteliklerinin değişimi sonucu oluşan değişim (update) paketleridir. En çok karşılaşılan paket türleri bunlar olduğu için tez kapsamında bu tür iletişimler için çözüm yaklaşımları anlatılmış ve bu yaklaşımların benzetimi gerçekleştirilerek analizleri yapılmıştır. Bu iletişim türlerinden üye olma ve yayınlama iletişimlerini deklarasyon yönetimi servisi ele alır. Değişim (update) ile ilgili iletişimleri ise nesne yönetimi servisi yönetmektedir [13,18].

Bu iletişim türlerinin benzetim sırasında ilerleyiş şekli izlendiğinde, koşut işlemdeki toplu iletişim işlemlerine (collective communication) çok benzediği görülmektedir. [14] numaralı çalışmadaki toplu iletişimler için önerilen yöntem, tez kapsamında inceleyeceğimiz iletişim örüntülerinin çıkış noktası olmuştur.

Bu bölümde, öncelikle tez çalışması kapsamında analizi yapılan deklarasyon yönetimi ve nesne yönetimi servisleri açıklanacak. Sonra işlemcilerin veri yapıları incelenecek, bu alt başlıkta önerilen yöntemlerin üzerinde işletildiği torus yapısı ve HLA'in fotonik ağlarla torus yapısı üzerinde tasarımı anlatılacaktır. Ardından, veri iletişimi için kullanılacak iletişim örüntüleri [7,14] ve hasır yöntemi açıklanacaktır.

Üye olma ve değişiklik (update) yayınlama paketlerinin önerilen iletişim örüntüsü ile nasıl iletildiği açıklanıp, son olarak ise veri iletişimi için önerilen örüntü ve hasır yapılarının birlikte kullanımı gösterilecektir.



Şekil 5.1 Deklarasyon Yönetimi Servisi

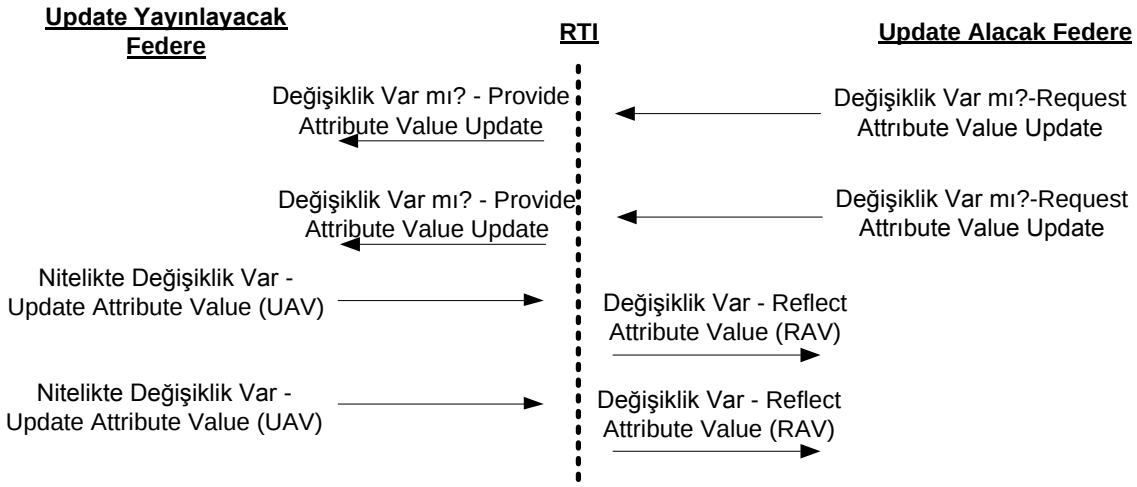
5.1. Deklarasyon Yönetimi Servisi

Deklarasyon yönetimi servisi, federelerin yönettiği nesnelerin üzerlerindeki niteliklerini yayınlamalarını ve başka nesnelerin niteliklerine üye olmalarını sağlar.

Şekil 5.1'de deklarasyon yönetimi için nitelik yayınlayacak ve niteliğe üye olacak federelerin RTI ile olan iletişimleri gösterilmiştir [18]. Şekilde niteliklerini yayınlayacak federe kendisinin yönettiği niteliklerden hangilerini yayınlayacağını öncelikle RTI'a bildirir. Ardından bu niteliklere üye olmak isteyen federeler üye olduklarını RTI'a iletirler. Buraya kadar olan kısımda federeler sadece birbirlerinin hangi niteliklerine üye olduklarını belirtirler. Sonraki aşamalarda niteliği yayınlanan sınıflardan bir nesne oluşturulur ve sisteme kaydedilir, üye olan federeler ise bu nesneleri keşfeder.

Her federe yayınlama ve üye olma işlemlerini RTI'a bildirir. RTI ise bu istekleri tüm diğer federelere bildirir. Burda görüleceği gibi her yerden-

her yere (all-to-all) bir iletişim vardır. Yayınlama ve üye olma istekleri ileride anlatılacak örüntülerin her yerden-her yere iletişimi sağlamaları sayesinde gerçekleştirilecektir.



Şekil 5.2 Nesne Yönetimi Servisi

5.2. Nesne Yönetimi Servisi

Nesne yönetimi servisi, federelerin yayınladıkları niteliklerindeki değişiklilerin duyurulmasını ve federelerin üye oldukları niteliklerin değişimlerini almalarını sağlar.

Bu serviste iki sebeple veri iletimi olabilir. İlkinde, değişiklik (update) alacak federe, bu federe aslında değişiklik beklediği federenin ilgili özelliğine üye olmuştur, kendisi için ilgili nitelikteki değişiklikerin önemli olduğunu düşündüğü durumlarda herhangi bir değişiklik olup olmadığını RTI'a sorar. RTI da gelen isteği değişikliği yayınlayacak federeye iletir. Değişikliği yayınlayacak federe, eğer değişiklik varsa onla ilgili bir paket gönderir. RTI da bu paketi değişikliği isteyen federeye iletir. İkinci veri iletimi sebebinde, değişikliği yayınlayacak federe herhangi bir istek gelmeden değişikliği RTI'a bildirir. RTI da gelen değişikliği önceden bu niteliğe üye olmuş, değişiklik bekleyen federelere gönderir. Şekil 5.2'de nesne yönetimi servisinde federelerin RTI ile olan iletişimleri gösterilmiştir [18].

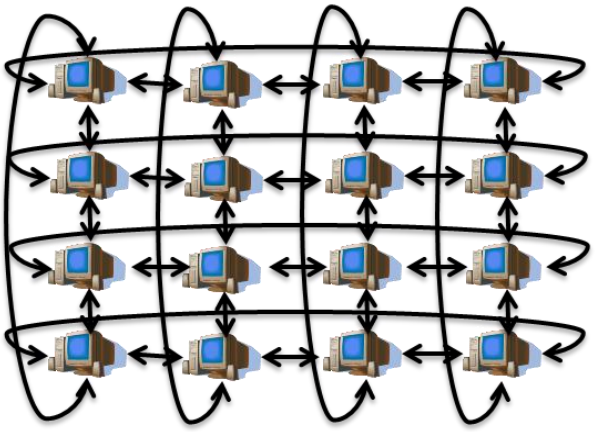
Bu servisde değişikliği tüm federeler göndermektedir. Değişiklikleri alan RTI ise sadece ilgili değişikliğin gerçekleştiği niteliğe üye olan federelere paketleri iletir. Yani parçalı her yerden-her yere bir trafik vardır. Bu servisin yönettiği veri iletişimleri de ileride açıklanacak örüntünün her yerden-her yere iletişimi sayesinde gerçekleştirilebilmektedir.

5.3. Önerilen Fotonik Ağ Mimarisi

Bu başlıkta işlemci üzerindeki çekirdeklerin nasıl bir ağ yapısı ile tasarlanacağı ve önerilen işlemci yapısı incelenecektir. Öncelikle iletişim örüntülerinin üzerinde işletileceği torus yapısı anlatılacak. Ardından HLA'de federelerin ve RTI'ın nasıl çekirdekler üzerinde dağıtıldığı gösterilecektir.

5.3.1. Torus Yapısı

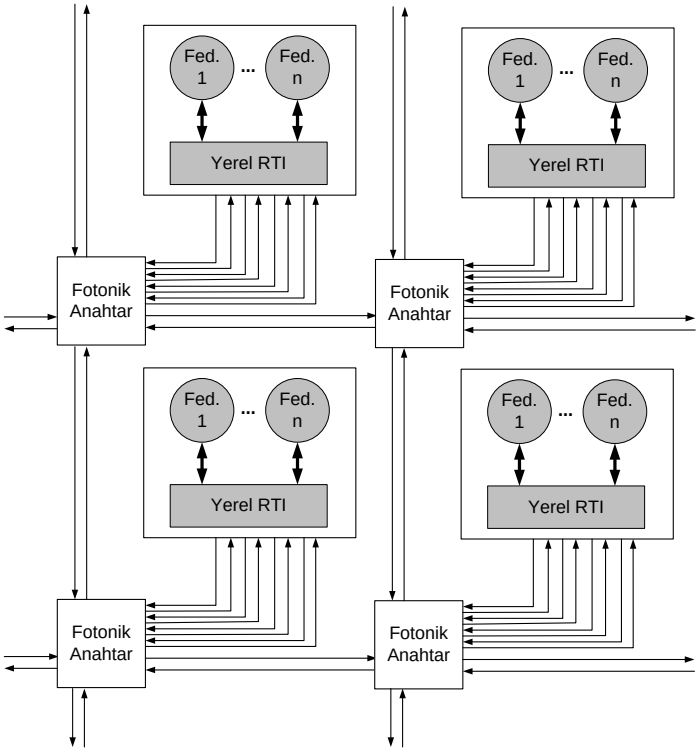
Torus yapılarında her bir işlemci 4 tarafındaki işlemcilerle iletişim halinde olacaktır ve kenarlardaki işlemciler kendi hizalarında kendilerine uzak kenardaki işlemcilerle de bağlantıya sahiptir [15,16]. Bu yapıda tüm yollar iki yönlü düşünülmüştür. Bu sayede karşılıklı iki işlemci veri alış-verişi yapabilmektedir. Şekil 5.3'de birbirlerine torus yapısıyla bağlı bilgisayarlar görülmektedir.



Şekil 5.3 Torus Yapısı

5.3.2. Önerilen Yapı

Şekil 5.4'de tez kapsamında önerilen işlemci yapısının bir kesiti örneklendirilmiştir [7]. Tasarlanan yapıda, şekilde de görülebileceği gibi her bir çekirdekte n tane federe ve bunların kendi aralarındaki iletişimlerini yöneten yerel RTI'lar vardır. Aslında bu tez kapsamında yapılacak olan yerel RTI'lar arasındaki iletişimin, ileride anlatılacak iletişim örüntüleri ile yapıldığı durumda, ne kadar etkin çalıştığını hesaplamaktır.

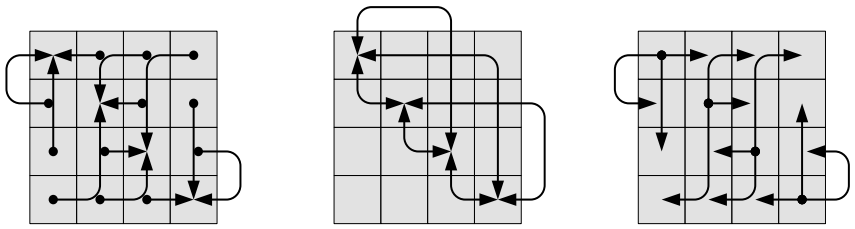


Şekil 5.4 Federeler, yerel RTI'lar ve işlemci ağ yapısından bir kesit

5.4. İletişim Örüntüleri

Önerilen örüntü, şekil 5.5'de verilen iletişim yollarını temel alarak gerçekleştirilecektir. 4x4'lük bir torus yapısı üzerinde örneklendirilecek örüntü, ileride anlatılacak farklı boyutlardaki örüntülerle birlikte kullanılarak, herhangi bir NxN'lik torus içinde işletilebilmektedir. Bu örüntünün her bir aşamasında, fotonik anahtarların içindeki mikro halkaların aktiflik-pasiflik durumları [7] önceden ayarlanır. Bu sayede

aralarında iletişim kuracak işlemciler arasındaki veri yolları birbirleriyle çakışmayacak şekilde ayarlanabilmektedir. Örneğin, şekil 5.5-b'de ilk satırdaki 3. işlemcinin veri dağıtımını sağlayan anahtar şekil 4.2-c'deki gibi kurulursa veriler birbirlerini engellemeden ilerleyebilmektedir. Diğer anahtarlar da bu şekilde konfigüre edilebilir. Bu sayede verilerin iletimi sırasında oluşabilecek beklemeler mümkün olduğunca en aza indirilmiş olmaktadır. Örüntünün adımları arasında bariyerler olacaktır. Bu bariyerler sayesinde bir örüntü adımı bitmeden sonraki adıma geçilemeyecektir. Bariyerlerin bir gerekçesi de sonraki adımın veri yollarının ayarlanması için fotonik anahtarların yapılandırılabilmesidir.



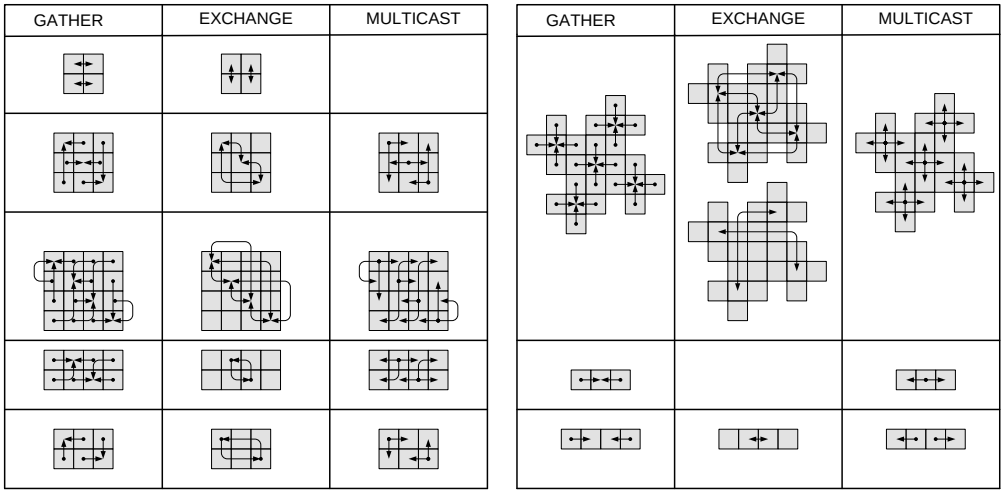
Şekil 5.5 Veri iletişimi için veri yollarının izleyeceği yollar. (a) Gather-Veri toplama işlemi (b) Exchange-Veri değişim işlemi (c) Multicast-Veri dağıtım işlemi

Bu örüntü sayesinde verilerin iletildiği yollar/rotalar kesişmeden iletişim gerçekleştirilebilmektedir. Örüntünün her bir adımında veri yolları birbirini engellemeyecek şekilde ayarlanmaktadır. Bu şekilde olduğu için veri yollarının hangi paket tarafından kullanılacağı problemi ortadan kalkmış olmaktadır. Her bir paket başka hiçbir akıllı sisteme (yönlendirici, paketlerin analiz edilmesi vs.) gereksinim duymadan gideceği işlemciye iletilebilmektedir. Bu manyetik halkaların, örüntü adımları arasında rotaların birbirleriyle kesişmeyecek şekilde kurulması sayesinde olur.

Şimdi herhangi bir işlemciden, sistemdeki herhangi başka bir işlemciye verileri, şekil 5.5'deki örüntüleri kullanarak nasıl aktarabileceğimize bakalım. Aslında burada gösterilecek örüntü her yerden-her yere (all-to-all) [19,20] iletişim için uyarlanmış bir örüntüdür [7,14]. Öncelikle

işlemciler 4 gruba ayrılmaktadır, diyagonal üzerindeki işlemciler *merkezi işlemciler* olarak işleme sokulacaktır. Her işlemci, şekil 5.5-a'da gösterildiği gibi gönderilecek verilerini diyagonal üzerindeki kendi grubu ile ilişkili merkezi işlemciye göndermektedir (Gather-Toplama işlemi). Bu verileri alan merkezi işlemciler şekil 5.5-b'deki gibi diğer diyagonal üzerindeki işlemcilerle veri alışverişinde bulunmaktadır (Exchange-Değişim işlemi). Burada tüm veriler diğer merkezi işlemcilere gönderilmez, sadece verinin gideceği işlemcinin bulunduğu grubun merkezi işlemcisine gönderilir. Üçüncü aşamada ise alınan bu veriler ilgili hedef işlemcilere dağıtılır (Şekil 5.5-c Multicast-Dağıtım işlemi).

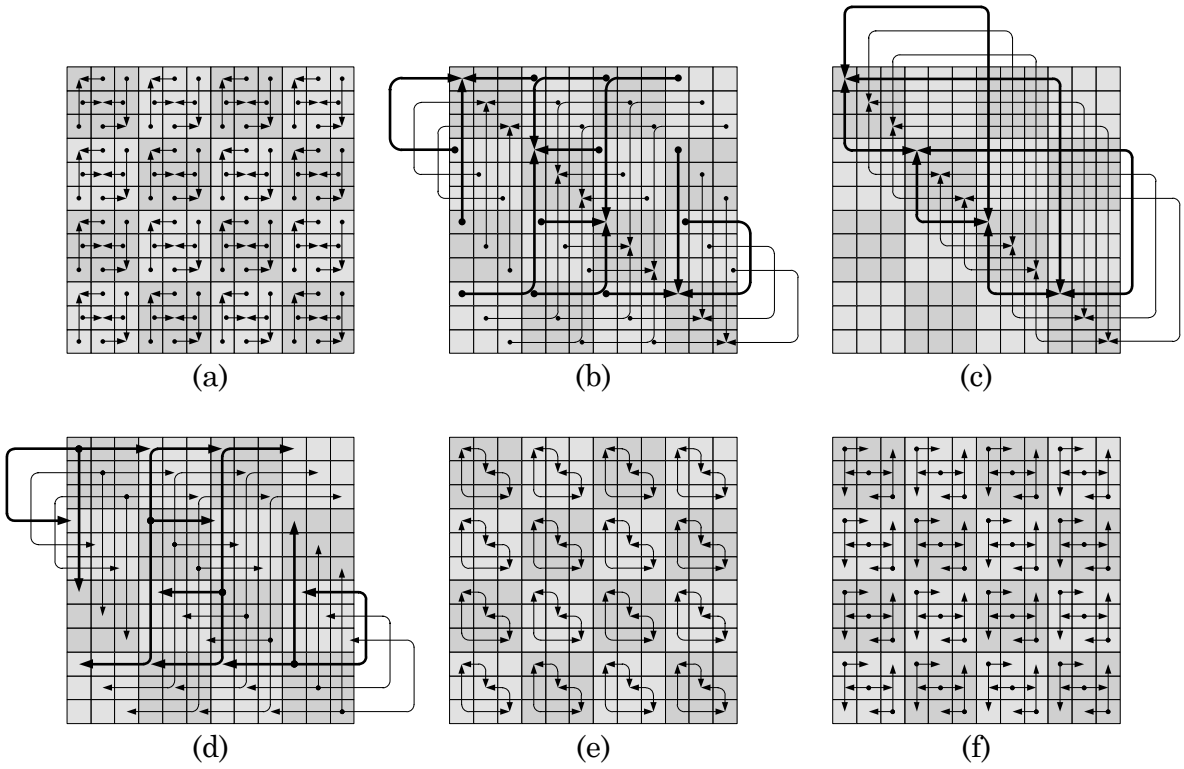
Yukarıda verilen 4x4'lük örüntü herhangi bir $4^n x 4^{n\prime}$ lik torus yapısı üzerinde işletilebilmektedir. Diğer boyutlardaki toruslar için şekil 5.6'da verilen örüntülerin birkaçı bir araya getirilerek işletilebilir [14].



Şekil 5.6 2x2, 3x3, 4x4, 2x4, 2x3, 5x5, 1x3 ve 1x4 torus yapıları için örüntülerde gather, exchange ve multicast işlemleri

Örneğin, 12x12'lik bir torus 4x4 ve 3x3'lük iki örüntünün birleşimi şeklinde işletilebilir. Şekil 5.7 de 12x12'lik bir torusun 3x3 ve 4x4'lük örüntülerin birleştirilmesi ile nasıl işleme sokulduğu adım adım gösterilmiştir. Şekil 5.7-a'da 3x3'lük küçük toruslar için verileri merkezi işlemcilere toplama (gather) işlemi yapılmıştır. Şekil 5.7-b'de ise 4x4'lük örüntünün gather işlemi yapılmıştır. Burada her bir 3x32lük işlemci gruplarını 4x4'lük örüntüde bir işlemci gibi düşünebilirsiniz, aslında

4x4'lük örüntü özyineli olarak büyütülmüş oluyor. Şekil 5.7-c'de genişletilmiş 4x4'lük örüntünün exchange işlemi gerçekleştirilmektedir, bu işlem sayesinde merkezi işlemciler arasında gerekli veri dağıtımı yapılmıştır. Şekil 5.7-d ise 4x4'lük örüntünün son adımı olan verilerin ilgili 3x3'lük bloklara dağıtılması işlemi yapılmaktadır. Artık özyineli işlemlerde bir alt aşama olan 3x3'lük örüntünün işletimine devam edilecektir. Artık kendisi ile ilgili tüm verileri merkezi işlemcilerinde toplayan 3x3'lük gruplardaki işlemciler, şekil 5.7-e'de verileri ilgili işlemcilere iletmek üzere merkezi işlemciler arasında exchange işlemi yapılmaktadır. Son işlem olan f'de ise 3x3'lük blokların merkezi işlemcileri, verileri ilgili işlemcilere dağıtmaktadır.

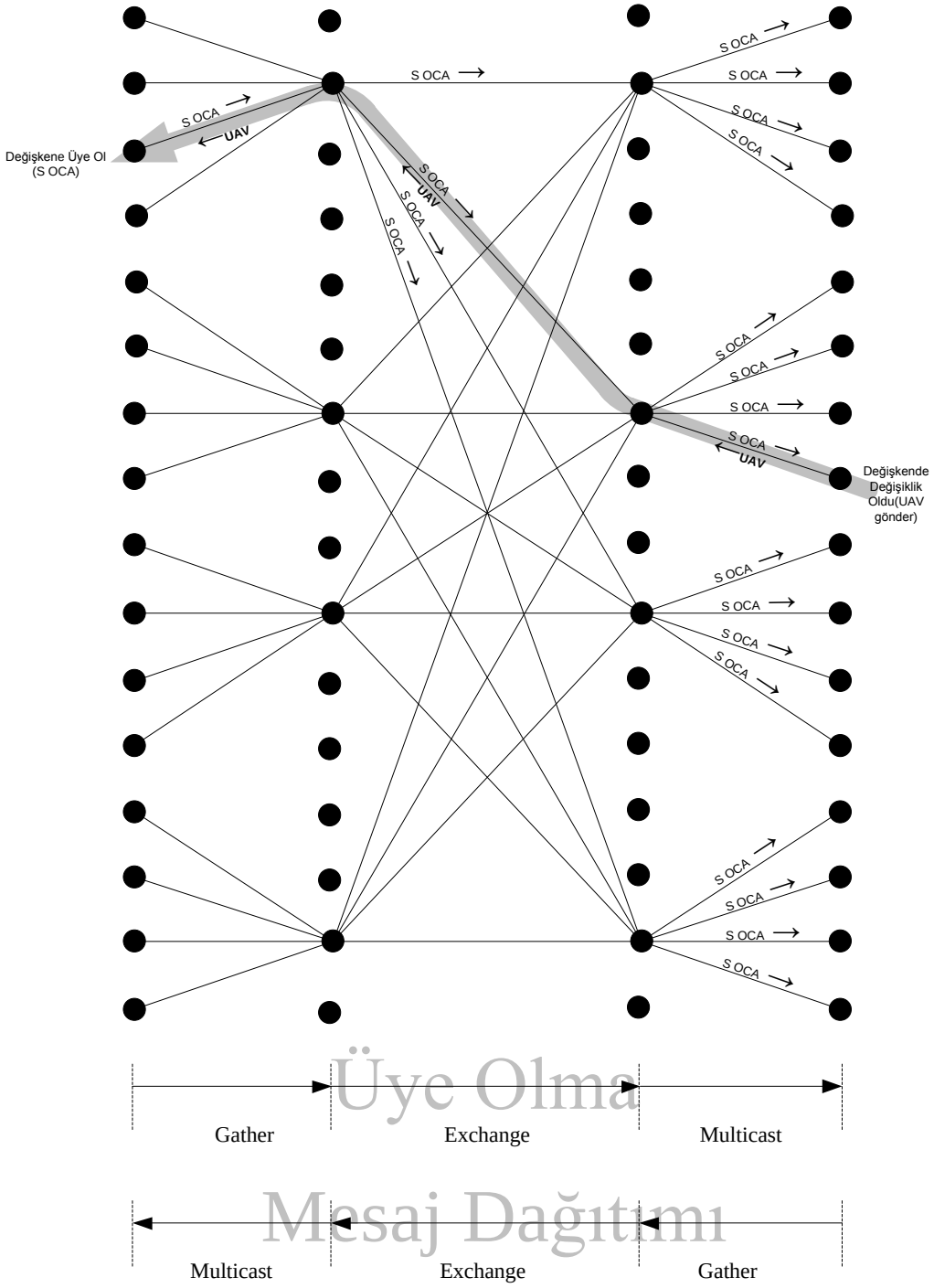


Şekil 5.7 12x12'lik bir torus için örüntünün aşamaları

5.5. Örüntü Üzerinde Üye Olma ve Değişiklik Gönderme

Bu alt başlıkta federeler arasında üye olma ve değişiklik (update) paketlerinin ilerleme şekli örüntü üzerinden işletilecektir. Şekil 5.8'de

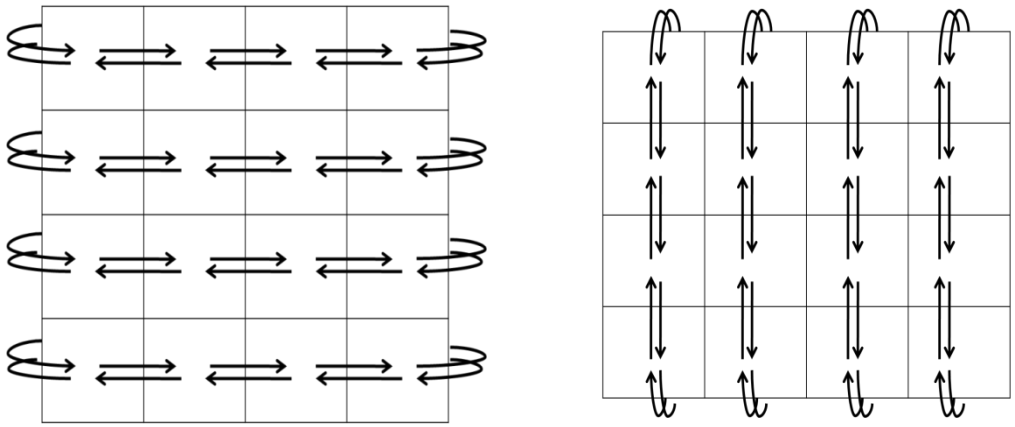
her bir aşamada, aynı satırdaki noktalar aynı işlemciler olarak düşünelebilir. Bunlar sadece farklı zaman dilimlerinde işlemciler arasındaki iletişimi göstermek için mantıksal olarak çizilmiştir. 2'nci ve 3'üncü sütundaki yolların toplandığı işlemciler köşegen üzerindeki merkezi işlemcilerdir. Bu örnekte en üstten 3'üncü sıradaki işlemci *S OCA- Subscribe Object Class Attribute*-Nesne Sınıfı Özelliğine Üye Ol isteğini taşıyan bir paket yaymaktadır. Bu paket *S Subscribe* yani üye olmak manasını taşımaktadır. Subscribe paketleri nesnelerin belli bir niteliğine üye olma paketleridir. Şekildeki paket, üstten 3'üncü işlemcinin içindeki federelerden birinin 8'inci işlemcideki federelerden birine ait nesnenin niteliğine üye olduğunu göstermektedir. Subscribe paketleri broadcast olarak tüm işlemcilere iletilmektedir; çünkü dağıtık benzetimlerde işlemcilerin yönettiği federeler başka işlemcilere taşınabilir veya başka bir işlemcide üye olunan nesnelerden bulunabilir. Bu paketler geçtikleri yollar üzerindeki işlemcilere bir iz bırakmaktadır, bu iz değişiklik (update) paketlerinin gideceği yerleri belirlemesine yarayacaktır. Sağ taraftan başlayan UAV–Update Attribute Value başlıklı paket ise çıktığı işlemcide üye olunan niteliğin değiştiğini belirtmektedir. Update paketi yayınlandıktan sonra üye olma işleminin tersi yönde bir yol izlemektedir. Eğer update paketinin ilgili olduğu nitelik, önceden yayınlanmış subscribe paketlerinin başkalarıyla da ilgiliyse, ilgili olduğu bu paketlerin izleri takip edilerek üye olan tüm işlemcilere update paketi iletilmektedir.



Şekil 5.8 4x4'lik torus üzerinde üye olma ve mesaj dağıtım işlemlerinin modellenmesi

5.6. Hasır Yöntemi

Her yerden-her yere (all-to-all) iletişimleri gerçekleştirebileceğimiz bir diğer yöntem hasır yöntemidir. Bu yöntem boyut sıralı yönlendirme (dimension order routing) olarak da bilinmektedir. Hasır yöntemi özyineli bir algoritma değildir ve torus yapısı üzerinde gerçekleştirilebilecek bir yöntemdir. Aslında yoğun her yerden-her yere iletişimleri gerçekleştirmek için kullanılan etkin yöntemlerden biridir. Bu yöntem için 4x4'lük torus yapısı üzerinde örneklendirilmesi şekil 5.9'da verilmiştir. Sırasıyla şekil 5.9-a'daki veri iletimleri 2 kere ve şekil 5.9-b'deki veri iletimleri 2 kere yapılarak tüm işlemcilerden tüm işlemcilere veri iletimi sağlanabilmektedir. Burada 5.9-a'da paketler yatay düzlemde gidecekleri işlemcinin bulunduğu sütuna kadar işlemciler üzerinden ilerlemektedir. Sonra dikey düzlemde asıl gideceği işlemciye kadar ilerlemektedir. Her iki düzlemde de paketler iki yönlü ilerleyebildiğinden düzlemdeki işlemci sayısının yarısı kadar ilerleme paketlerin tüm işlemcilere dağıtılması için yeterli olacaktır.



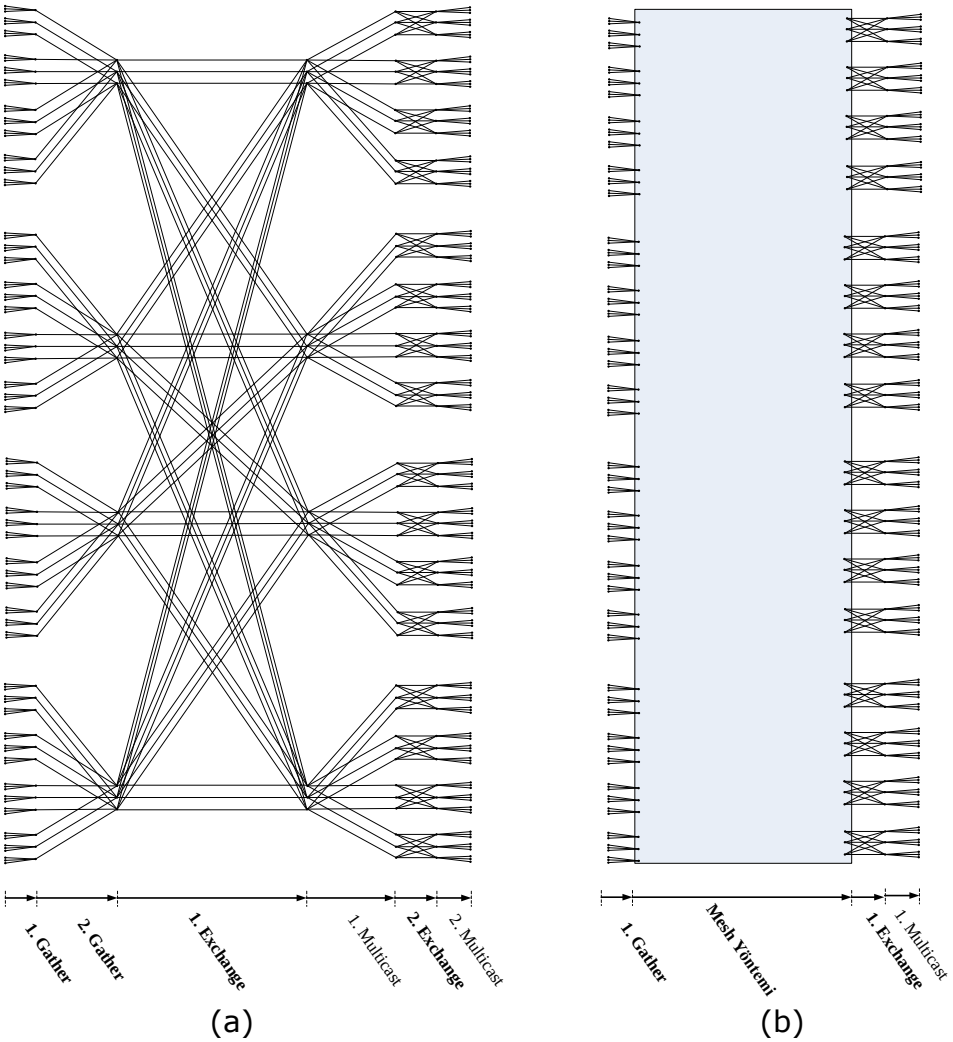
(a) (b)

Şekil 5.9 Hasır yöntemi için işlem adımları

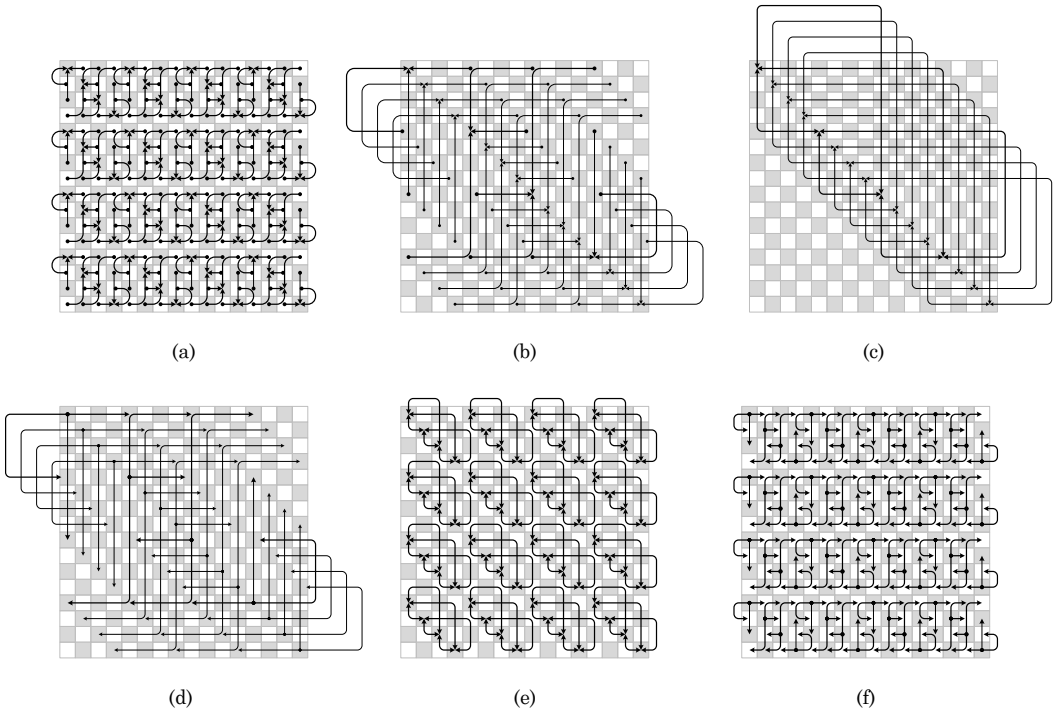
5.7. Örüntü-Hasır Birlikte Veri İletişimi

5.4 bölümünde anlatılan iletişim örüntüsü ile hasır yapısı birbirleriyle birlikte çalışabilecek yöntemlerdir. All-to-all iletişim büyük torus yapılarında bu iki yöntemin birlikte kullanılması ile de gerçekleştirilebilir. İki yöntemin de birbirlerine göre avantajlarından bu şekilde yararlanılabilir.

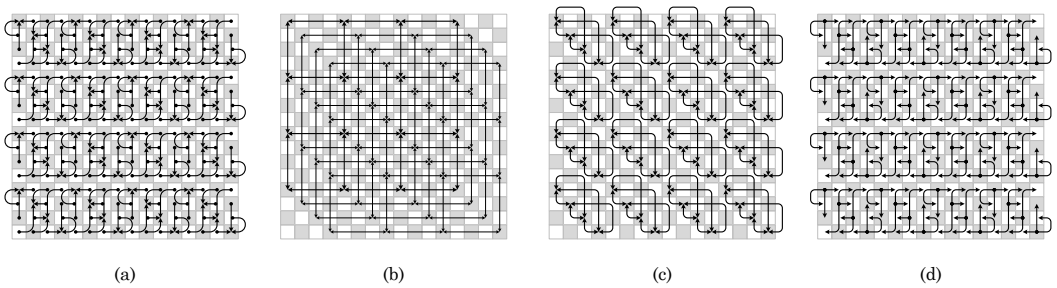
Bu çalışmada, işlemler iletişim örüntüsü gibi başlatılıp özyineli olarak ilerlerken bir yerde hasır yöntemine geçilecektir. Şekil 5.10'da temsili olarak tamamen örüntünün işletildiği ve araya hasır yönteminin koyulduğu durumlar gösterilmiştir.



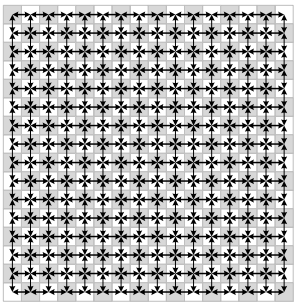
Şekil 5.10 Veri dağıtımı için sadece örüntü ve örüntü-hasır yöntemlerinin beraber kullanıldığı durumlar



Şekil 5.11 16x16 işlemciye sahip torus yapısı için tamamen örüntünün işletildiği durumlarda işlemciler arasındaki veri iletimi adımları.



Şekil 5.12 16x16 işlemciye sahip torus yapısı için bir adım örüntünün bir adım hasır yapısının işletildiği durumlarda işlemciler arasındaki veri iletimi adımları.



Şekil 5.13 16x16 işlemciye sahip torus yapısı için sadece hasır yapısının işletildiği durumlarda işlemciler arasındaki veri iletimi yolları.

Şekil 5.11, 5.12 ve 5.13'de 16x16 boyutlarındaki torus yapısı için tamamen örüntünün, bir adım örüntü bir adım hasır ve sadece hasır yapısının işletildiği durumlarda veri yollarının nasıl olacağı çizilmiştir. Tamamen örüntünün olduğu şekil 5.11'de özyineli olarak iki tane içi içe 4x4'lük örüntü işletilmiştir. Şekil 5.12'de işlem adımları 4x4'lük örüntü ile başlamış sonra şekil 5.12-b'de hasır yapısına geçilmiş ve tekrar 4x4'lük örüntüye devam edilmiştir. Burada 5.12-b şeklindeki iletişim 4 kere yapıldıktan sonra 5.12-c ye geçiş yapılmaktadır. Şekil 5.13'de ise sadece hasır yapısının kullanıldığı durumda veri yolları görülmektedir. Bu iletişim ise 16 kere yapılarak her yerden-her yere iletişim gerçekleştirilebilmektedir.

6. ANALİZ

Bu bölümde, nesne yönetimi servisi için, önceki bölümde açıklanan iletişim örüntüsünün ve hasır yönteminin etkinliği ölçülecektir. Öncelikle iletişim masraflarının hesaplanacağı formül açıklanacak, ardından örnek bir trafik, örüntü üzerinde işletilerek formül yardımıyla yaklaşık bir işlem masrafı hesaplanacaktır. Hasır yönteminin ne için kullanıldığından bahsedilecek son olarak da nesne yönetimi servisinin modellendiği deneylerin sonuçları grafikler üzerinde incelenecektir.

6.1. Formülasyon

Çekirdekler arasında veri iletimi sırasında geçen süreyi hesaplamak için 6.1 numaralı formül kullanılacaktır. Bir paketin bir çekirdekten diğer çekirdeğe iletilirken geçen süre iki parçada incelenebilir. Bunlardan birincisi hazırlık sürecidir, bu kısımda verinin iletilebilir anlamlı bir paket haline getirilmesi, verinin ilerleyeceği rotanın belirlenmesi gibi veri iletiminden önce yapılması gereken işler yapılır ve formülde $\boldsymbol{\alpha}$ olarak ifade edilmiştir. Veri iletiminin ikinci kısmı ise, iki işlemci arasında paketler iletilirken geçen süredir. Formülün ikinci kısmı bu geçen süreyi belirtmektedir. $\boldsymbol{l}$ veri paketinin boyutunu gösterirken $\boldsymbol{\tau}$ verininin iletildiği yol için belirlenmiş ve yolun veri iletimi hızını belirten bir sabittir.

$$\alpha + l * \tau$$

Formül 6.1 İki işlemci arasında veri iletimi için geçen süre

Formülde yer alan değişkenlenlerin birimlerini inceleyelim. $\boldsymbol{\alpha}$ değişkeni iletişime hazırlık için geçen süre olduğundan birimi $\boldsymbol{saniye}$ dir. $\boldsymbol{l}$ değişkeni iletilecek paketin boyunu ifade ettiğinden birimi $\boldsymbol{bayt}$, $\boldsymbol{\tau}$ değişkeni ise iletim yolunun bir $\boldsymbol{bayt}$ veriyi ne kadar sürede taşıyabildiğini belirttiği için birimi $\boldsymbol{saniye/bayt}$ olacaktır. Formül birimler

üzerinden yazıldığı zaman $\boldsymbol{sn} + \boldsymbol{bayt} * \frac{\boldsymbol{sn.}}{\boldsymbol{bayt}}$ şeklinde bir ifade karşımıza çıkıyor. Bu ifadenin sonucu da $\boldsymbol{saniye}$ olacaktır. Yani bu formülle hesaplanacak işlemlerin sonucu $\boldsymbol{saniye}$ cinsinden olacaktır.

6.2. İletişim Masrafı Hesaplamanın Modellenmesi

Örüntü nesne yönetimi servisi için kullanılmadan önce, normal bir veri trafiği ile nasıl çalıştığı ve bu trafik neticesinde nasıl bir masraf ortaya çıkardığı incelenecektir. Bunun için şekil 5.7'de verilen 12x12'lik torus yapısı üzerinde iletişim örüntüsünün işletimi adım adım ilerletilecek ve her bir adımın masrafı formül 6.1 kullanılarak hesaplanacaktır. Örneklendirilecek veri trafiği ise şu şekilde olacaktır: Her iki işlemciden birinin başka bir işlemciye mesaj yollamak istediği varsayılacak, yani örneğin mesaj yoğunluğu %50 oranında olacaktır. Örnekte örüntünün verimliliğininin ölçülebilmesi için α ve l değişkenlerine bir değer atanmayacak α ve l olarak hesaplanacaklardır. τ değeri ise örnekte ve sonraki analizler boyunca sabit alınacaktır.

12x12 torus yapısı (şekil 5.7) 5.4 bölümünde belirttildiği gibi 3x3 ve 4x4'lük örüntülerin birlikte kullanılması ile işletilebilir. Öncelikle torus yapısı 3x3 lük parçalara ayrılacak, bu 3x3 lük parçaların her biri bir elemanı olacak şekide tüm torus 4x4 olarak ayrılmış olacaktır. Bu işlemden sonra şekil 5.6'de belirttilen 3x3 ve 4x4'lük örüntüler sırasıyla özyineli olarak işletilecektir. Şekil 5.7-a, 5.7-e ve 5.7-f özyinelemenin ilk katmanını gösterirken, 5.7-b, 5.7-c ve 5.7-d ikinci katmanını göstermektedir. Şekil 5.7-a'da 3x3'lük bir toplama (gather) işlemi yapılmaktadır, 5.7-b'de 4x4'lük bir toplama (gather) işlemi, 5.7-c'de 4x4'lük örüntü için değişim (exchange) işlemi, 5.7-d'de 4x4'lük örüntünün dağıtım (multicast) işlemi yapılmaktadır. 5.7-e'de tekrar 3x3'lük örüntüye dönülerek bu örüntünün değişim (exchange) işlemi, 5.7-f'de ise dağıtım (multicast) işlemi yapılmaktadır. Bundan sonra aşama aşama işlem masrafları hesaplanacaktır.

a- Her 3x3'lük grubun verileri, kendi içlerindeki merkezi işlemciler (diyagonel üzerindeki işlemciler) üzerinde toplanacaktır. Her iki işlemciden biri veri gönderme işlemini yaptığından, bu aşamanın masrafı paketlerin hazırlanma süresi olan $\boldsymbol{\alpha}$ ve her bir yoldan sadece bir paket gideceğinden $\boldsymbol{l * \tau}$ nin toplamı olacaktır. $(1 * \alpha + 1 * l * \tau)$

b- Bu aşamada 3x3'lük parçaların merkezi işlemcileri kendi aralarında 4x4'lük bir örüntü kurmaktadır. Burada birbiriyle kesişmeyen, paralel olarak işletilen 3 tane 4x4'lük örüntü olacaktır. 3x3'lük parçaların her bir merkezi işlemcisi kendi hizasındaki diğer 3x3'lük parçaların merkezi işlemcileri ile aynı 4x4'lük örüntü içinde olacaktır. Bu aşamanın masrafını şu şekilde hesaplayabiliriz; öncelikle paketlerin hazırlanması süreci olan α olacaktır. Bir önceki işlemde merkezi işlemcilerde bir veya iki paket toplanmıştı. Bu aşamada bu paketler dağıtılacağı için en kötü durumu göz önüne alıp, iki paketinde aynı yere gideceğini varsayalım, bu durumda veri iletimi için $2 * l * \tau$ kadar süre geçecektir. $(1 * \alpha + 2 * l * \tau)$

c- Bu aşamada 4x4'lük örüntünün değişim (exchange) işlemi yapılacaktır. Burada yine paketlerin hazırlanması için geçen süre olan α en başta olacaktır. Formülün ikinci kısmını ölçmek aslında hangi paketin hangi işlemciye gittiğini bilmeden imkansızdır, bunun için ortalama bir değer alınacak. En kötü durumda, bir önceki adımda buraya paket gönderen işlemcilerden ikişer paket gelebilir, yani en fazla 8 paket gelebilir. En iyi durum ise birer paket gelme durumudur. Bu durumda 4 tane paket gelir. Bu aşamada 4 ile 8 in ortalaması olan 6 paket olduğu varsayılacaktır. Sonuç olarak, $6 * l * \tau$ kadar zaman geçecektir. $(1 * \alpha + 6 * l * \tau)$

d- 4x4'lük örüntünün son kısmı olan dağıtım (multicast) işlemi bu aşamada olacaktır. Bu aşamada 4x4'lük örüntünün merkezi işlemcileri, 4x4'lük örüntü için normal işlemciler, 3x3'lük örüntü için merkezi işlemciler olan işlemcilere paketleri gönderecektir.

Burada bir önceki adımla benzer olarak hangi işlemcilere veri gideceği bilinmeden tam bir hesaplama yapılamaz. Yine ortalama bir değer hesaplanacak. Bir önceki adımdan ortalama değer olan 6 paket geldiği düşünüldüğü durumda bir yoldan en fazla 6 en az 0 paket iletilecektir. Yani bu aşama için bir yoldan ortalama 3 paket geçtiği varsayılacak. Veri iletim masrafı, $3 * l * \tau$ olacaktır. En başta paketlerin hazırlanma kısmı olan α da eklendiği zaman bu aşama için toplam masraf $(1 * \alpha + 3 * l * \tau)$ olacaktır.

e- Bu bölümde 3x3'lük örüntünün değişim (exchange) kısmı yapılacaktır. 3x3'lük işlemci gruplarının merkezi işlemcileri arasında, sonraki aşamada asıl hedeflerine iletilmek üzere değişim (exchange) işlemi yapılacaktır. Burada işlemciler sonraki aşamada kendileri dahil dağıtacakları paketleri alacaklardır. Paketler %50 oranında oluşturulduğu için sonraki aşamaya 2 paket gönderecek işlemciler olacaktır. Bu durumda bu aşamanın veri iletim masrafı, $2 * l * \tau$ olacaktır. En başta paket hazırlanması süresi olan α da eklendiği zaman toplam masraf bu aşama için $(1 * \alpha + 2 * l * \tau)$ olacaktır.

f- Bu aşamada ise 3x3'lük merkezi işlemciler, paketleri asıl ulaşmaları gereken işlemcilere ulaştıracaklardır. Birden fazla paketin bir işlemciye gitmesi durumunu göz önüne almayacağız. Yani en baştan oluşturulan paketlerin her birinin farklı işlemcilere gönderilmek istendiğini varsayılacak. Bu durumda veri iletimi sırasında $1 * l * \tau$ kadar süre geçecektir. Paketlerin hazırlanması için geçen α süresiyle birlikte, bu aşamanın masrafı $(1 * \alpha + 1 * l * \tau)$ olacaktır.

Bu hesaplamalar yapıldıktan sonra iletişim örüntüsünün böyle bir trafik için ortalama ne kadar masraf çıkardığı incelenilecek. Tüm aşamalardaki masraflar toplandığı zaman, toplam masrafın $6 * \alpha + 15 * l * \tau$ olduğu görülecektir. Bu örüntüde paket yoğunluğu arttığı durumda formülün ilk kısmı olan $6 * \alpha$ değeri değişmeyecektir, sistemdeki işlemci sayısının

artması yani adım sayısının artması durumunda ise sınırlı bir artış olacaktır bundan dolayı α değerinin büyük olduğu senaryolarda örüntü etkin bir şekilde çalışacaktır.

6.3. Neden Hasır Yapısı?

Bu bölümde hasır yapısına neden ihtiyaç duyulduğu ve bu yapının veri iletişimine ne gibi katkıda bulunacağı incelenecektir.

Öncelikle örüntünün büyük boyutlardaki toruslar için nasıl bir olumsuz yanı olduğuna bakılacak. Örüntü özyineli olarak derinleştikçe merkezi işlemcilerin işlem yükü ciddi miktarda artacak ve bu adımlarda merkezi işlemcilerdeki paket sayısı katlanarak artacaktır. Bunun sonucu olarak veri iletim süresi de katlanarak artacaktır. Hasır yapısı ise örüntünün bir yerden sonra özyineli olarak daha derinlemesine inmesini engelleyecektir. Hasır yöntemi sayesinde paketler merkezi işlemcilerde toplanmak yerine dağıtık bir şekilde ilgili işlemcilere iletilir. Bu da paketlerin dar boğazda (merkezi işlemcilerde) toplanmasını engelleyecektir.

$4^n \times 4^n$'lik bir torus yapısında veri iletimi için hasır yönteminde toplam adım sayısı $(4^n + 4^n)/2$ olacaktır. Örüntü ile iletişimde ise $3 \times n$ adımlık bir işlem serisi yetecektir. n sayısı arttıkça bu iki yöntem arasındaki fark ciddi miktarda artmakdır. Bu da toplam taşınan paket sayısında aşırı bir artış olmamasına rağmen toplam masraftaki α miktarının artmasına sebebiyet vermektedir. Ama paketler dar boğazda birikmediği için n sayısının küçük değerlerinde hasır yöntemi daha mantıklı olabilir. Bu çalışmada yapılacak olan, işlemleri örüntü ile iletişim olacakmış gibi başlatıp özyineli olarak ilerlerken bir yerde hasır yöntemine geçerek özyinelemeyi kesip oluşabilecek dar boğazların önüne geçmek olacaktır. 5.7 bölümünde örüntü ile hasır yapısının nasıl birlikte çalışacağı ayrıntılı bir şekilde açıklanmıştır.

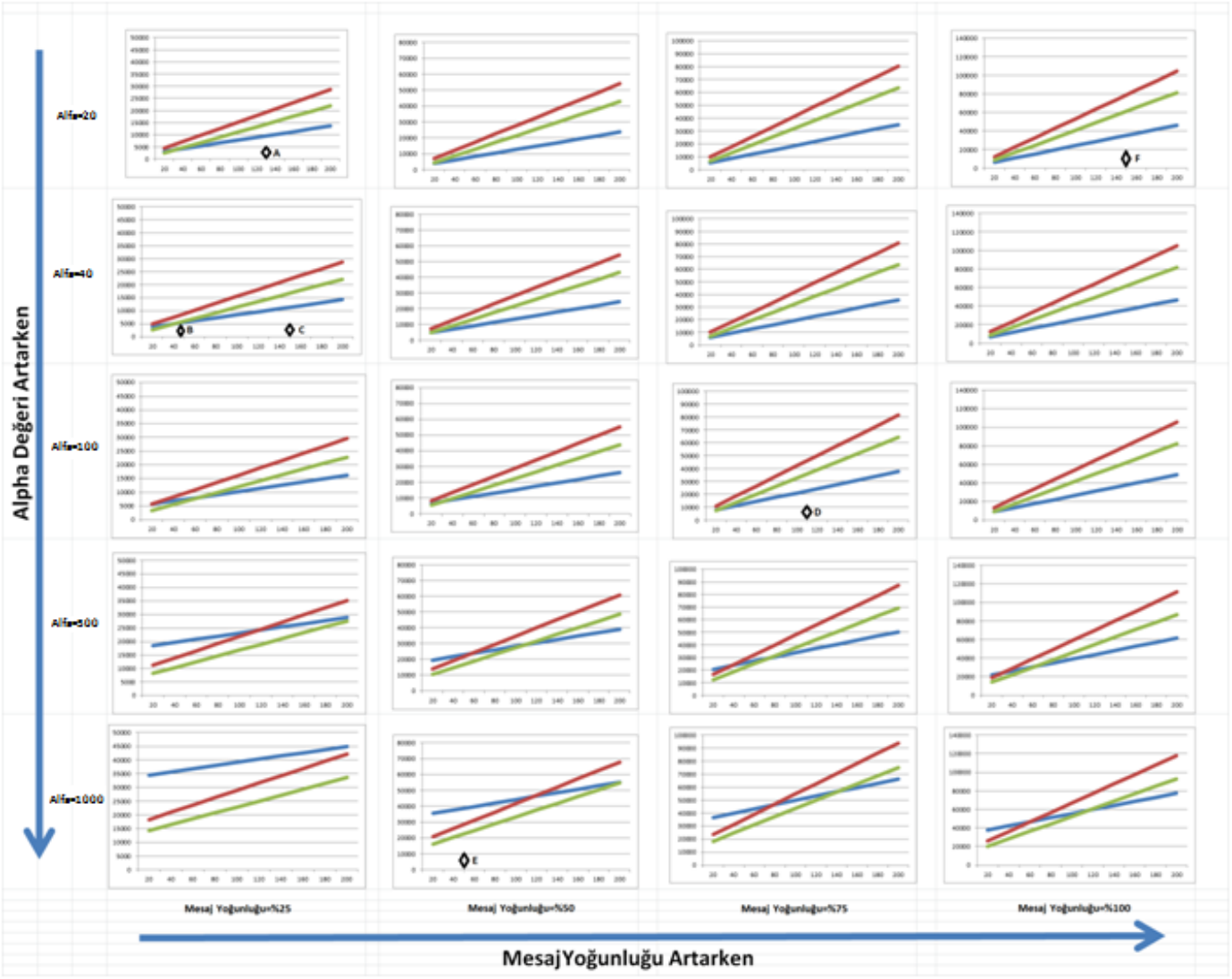
6.4. Deney Sonuçları

Bu tez kapsamında yapılan deneylerde, hem örüntünün etkinliği ölçülecek hem de hasır yapısının hangi test ortamlarında özyinelemenin kaçıncı basamağında girmesi gerektiği gösterilmiş olacaktır. Nesne yönetimininin de modellediği bu deneylerde her bir çekidekte bir federe olduğu ve tüm federelerin tüm diğer federelerin yayınladıkları nesnelerin niteliklerine üye oldukları düşünülmüştür.

6.4.1. Genel Değerlendirme

Şekil 6.1'de 16x16'lık bir torus yapısında veri iletimi için farklı deney ortamları hazırlanarak, bu deneylerin sonuçları gösterilmiştir. Şekildeki her bir grafikte alfa ve mesaj yoğunluğu sabit tutulup mesaj boyu değiştirilecek şekilde birçok deney düzeneği hazırlanmıştır. Genel olarak şekle bakıldığı zaman ise grafikler aşağı doğru indikçe alfa değerleri artmaktadır. Sağa doğru ilerledikçe ise mesaj yoğunluğu artmaktadır.

Grafiklerde mavi çizgiler, veri iletiminin tamanen hasır yapısıyla iletildiği; kırmızı çizgiler, sistemin önce örüntüyle başlayıp sonra hasır yapısına geçtiği test durumlarını; yeşil çizgiler, hiç hasır yapısı kullanılmadan sadece örüntünün kullanıldığı durumları göstermektedir.



Şekil 6.1 16x16 lık torus için alfa ve mesaj yoğunluğunun sabit tutulduğu durumlarda mesaj boyunun sonuçlara etkisini gösteren grafikler.

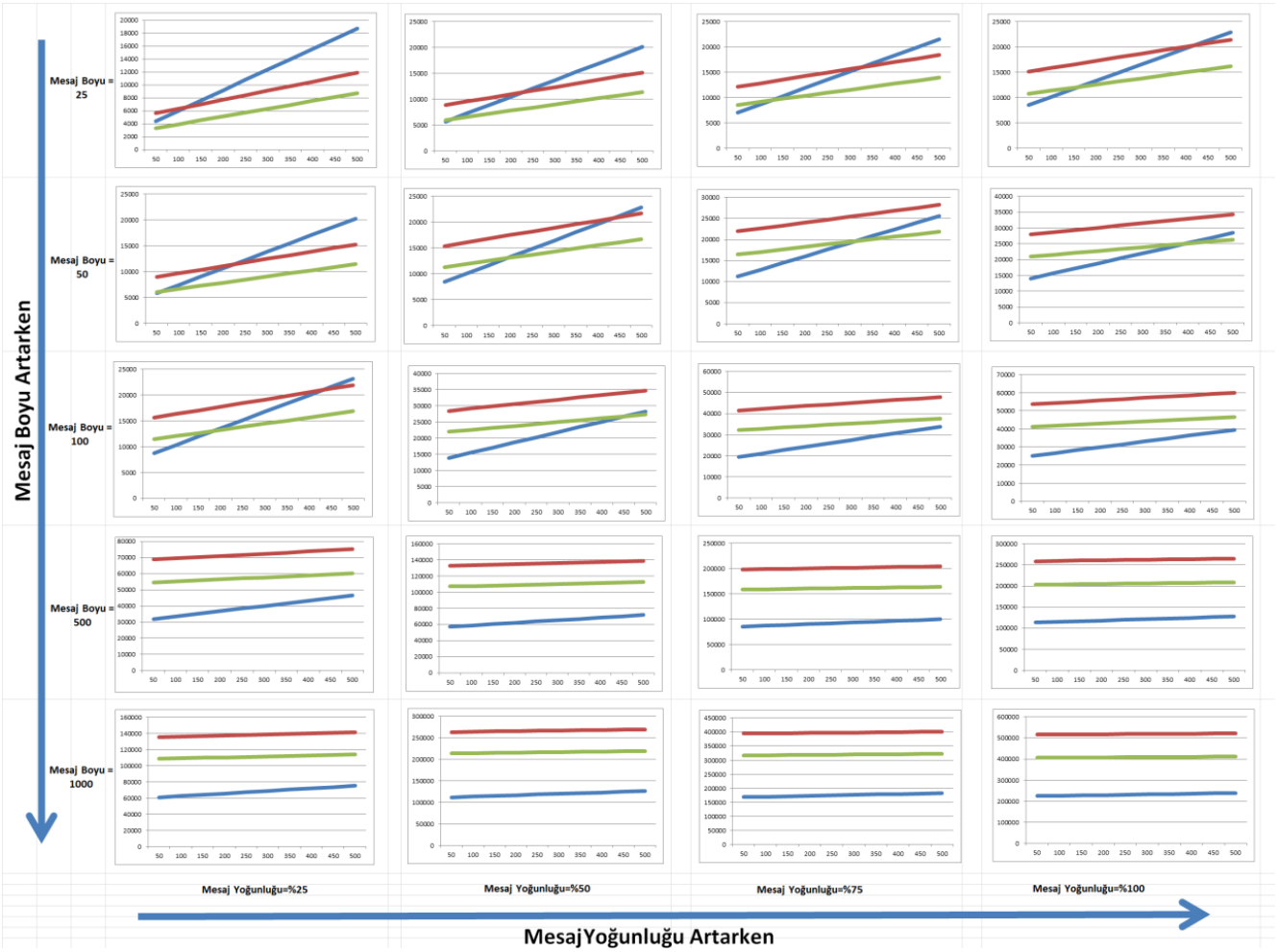
Bu deneylerde aslında tam olarak beklenen sonuçlar alınmıştır. Dikkat edildiği zaman ilk satırlardaki grafiklerde tamamen hasır yapısının kullanıldığı durumlar (mavi çizgiler) hep daha kısa zamanda veri iletimini sağlamaktadır. Hasır yapısının iletişim masrafında alfa sayılarının daha fazla olmasına rağmen böyle sonuçlar vermesinin sebebi alfa değerlerinin küçük olması ve iletişim masrafına çok etkisinin olmamasıdır. Grafiklerde görüleceği gibi alfa değerleri arttıkça hasır yapısının işlem masrafı gittikçe artmaktadır ve örüntünün dahil olduğu durumlardan daha verimsiz hale gelmektedir. Yani alfa değeri arttıkça örüntünün kullanılması daha uygun olmaktadır.

Şekil 6.1'de grafikler sağ tarafa doğru ilerledikçe mesaj yoğunluğu arttığı için örüntünün kullanıldığı yöntemlerde dar boğazın etkisi artmaktadır. Bu dar boğazlar sonucunda sistem daha yavaş çalışacaktır. Şeklin son satırında bu durum rahatlıkla görülebilir. Tamamen örüntünün kullanıldığı yöntemler (yeşil çizgi) ilk sutunda daha hızlı sonuçlar verirken, sağa doğru ilerldikçe hasır yönteminin kullanıldığı yöntemler (mavi çizgiler) daha hızlı sonuçlar vermeye başlamaktadır. Bunun sebebi hasır yönteminde paketler belli noktalarda birikmediği için işlem sayısı fazla olmasına rağmen dar boğazlar oluşmamaktadır. *Yani işlem yoğunluğu arttıkça hasır yöntemine doğru geçişler gerekmektedir.*

Şekil 6.2'de 16x16'lık bir torus yapısı üzerinde, çekirdeklerin veri dağıtım masrafları, farklı deney ortamları için hesaplanmıştır. Şekil 6.2'deki verilerin şekil 6.1'deki verilerden temel farkı; bir önceki şekilde grafiklerin her birinde alfa ve mesaj yoğunluğu sabit tutularak mesaj boyu artırılıyordu, burada ise her bir grafik için mesaj boyu ve mesaj yoğunluğu sabit tutularak alfa değeri arttırılmıştır. Bu grafiklerde, alfa değerinin artmasının veri iletişiminde kullanılacak yöntemi belirlemede nasıl etkili olduğu incelenecektir. Bununla birlikte, şekil genel olarak incelenerek alfanın aynı değerleri için mesaj yoğunluğunun ve mesaj boyunun masrafa etkisi gösterilecektir. Şekil 6.2'de grafikler aşağı doğru indikçe mesaj boyu artarken sağa doğru ilerlerken mesaj yoğunluğu artmaktadır.

Şekil 6.2'deki grafiklerde ilk satıra bakıldığı zaman yeşil (tamamen örüntü) çizgilerin daha avantajlı olduğu görülecektir. Ve aşağı doğru indikçe yeşil çizgiler mavi çizgilerin (tamamen hasır) üstüne doğru çıkmaktadır. Bunun sebebi, mesaj boyu arttıkça örüntünün çalıştığı durumlarda dar boğazların etkisinin artmasıdır. Bu etkiler arttıkça hasır yapısının kullanıldığı durumlar daha mantıklı duruma gelmektedir. Hasır yapısının kullanıldığı yöntemlerde toplam adım sayısının artması bile (adım sayısının artması, toplam masraftaki alfa sayılarının artmasına sebep olmaktadır) bir yerden sonra bu sonucun önüne geçememektedir.

Sonuç olarak mesaj boyu arttıkça örüntünün işlemlerdeki etkisini azaltıp hasır yapısının daha çok kullanıldığı yöntemlere geçilmelidir



Şekil 6.2 16x16 lık torus için mesaj boyu ve mesaj yoğunluğunun sabit tutulduğu durumlarda alfanın sonuçlara etkisini gösteren grafikler

6.4.2. Ayrıntılı Değerlendirme

Bu bölümde deney sonuçlarının ayrıntılı değerlendirmesi yapılacaktır. Alfa, mesaj yoğunluğu ve mesaj boyuna göre grafiklerin yorumlanması bu bölümde ayrıntılı bir şekilde değerlendirilecektir. Aslında mesajların çıkış ve varış çekirdeklerinin dağılımı bilinmeden herhangi bir yorum

yapmak doğru olmaz. Çünkü, örneğin sistemin genelinde mesaj yoğunluğu az olsa bile bazı bölgelerde bölgesel yoğunluklar olabilir, bu da örüntü yöntemi kullanılırken tüm sistemin yavaşlamasına sebebiyet verecektir. Bunun gibi nedenlerden bu bölümde yapılacak yorumlar genel itibariyle doğru olsa da bazı istisnai durumlarda yanlış olabilecektir.

Öncelikle alfa değerine göre grafikler incelenecek:

- Alfa değeri 100 ms'den küçük olduğu durumlarda, sadece hasır yöntemi hep daha hızlı çalışmaktadır.
- Alfa değeri 100 ms'den küçük olduğu durumlarda, 1 seviye örüntü 1 seviye hasır yönteminin kullanıldığı durumlar hep en yavaş sonucu vermektedir.
- Alfa değeri 500 ile 1000 ms aralığında olduğu durumlarda, mesaj boyu 350 bayttan büyükken sadece hasır yapısı en hızlı sonuçları vermektedir.

Mesaj boyuna göre grafiklerin incelenmesi:

- Mesaj boyu 100 baytın üstündeyken alfa değeri 500'ün altında olduğu durumlarda, hasır yöntemi hep daha hızlı çalışmaktadır.
- Mesaj boyunun 50 baytın altında olduğu durumlarda, mesaj yoğunluğu %75'in veya alfa değeri 400 ms'nin altındayken sadece örüntünün kullanıldığı durumlar en iyi sonuçları vermektedir.
- Mesaj boyunun 350 bayttan büyükken alfa değeri 1000 ms'nin altındaysa, sadece hasır yönteminin kullanıldığı yöntemler daha hızlı çalışmaktadır.

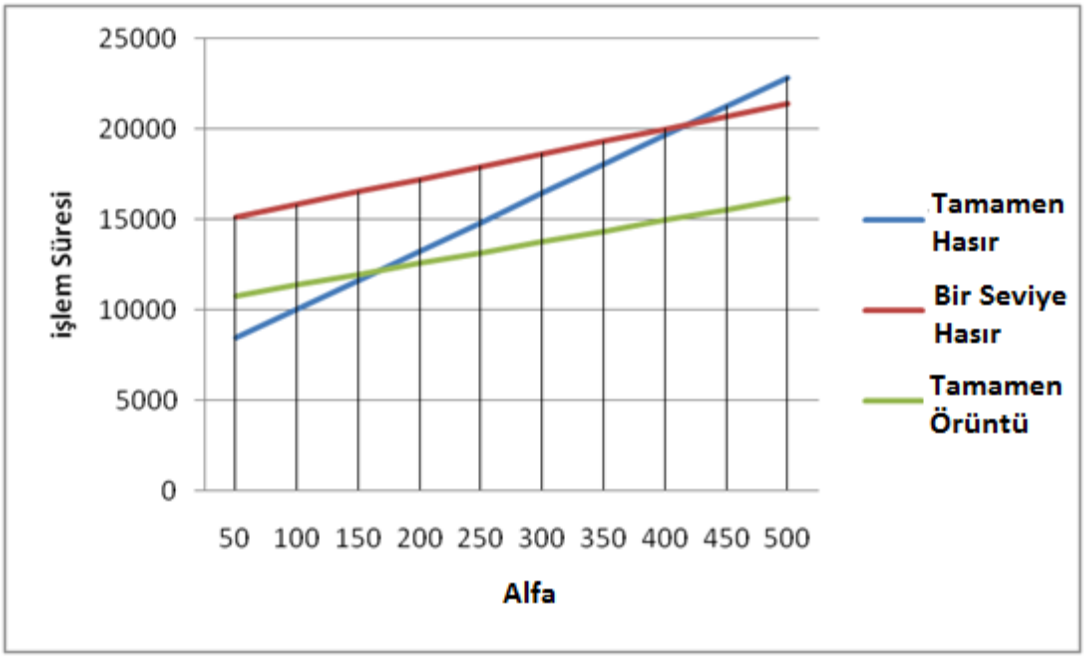
Mesaj yoğunluğuna göre grafiklerin incelenmesi:

- Mesaj yoğunluğu %50'ye kadar olduğu durumlarda;
    - Alfa değeri 100 ms'den az olduğu durumlarda, sadece hasır yönteminin daha hızlı sonuçlar verdiği görülmektedir.

- Mesaj boyunun 100 bayttan az olduğu durumlarda, tamamen örüntünün kullanıldığı yöntem hep daha iyi sonuçlar vermektedir.
- Alfa değeri 500 ms'nin üstündeyken, mesaj boyu 200 bayt olana kadar tamamen örüntünün kullanıldığı yöntem daha hızlı sonuçlar vermektedir.

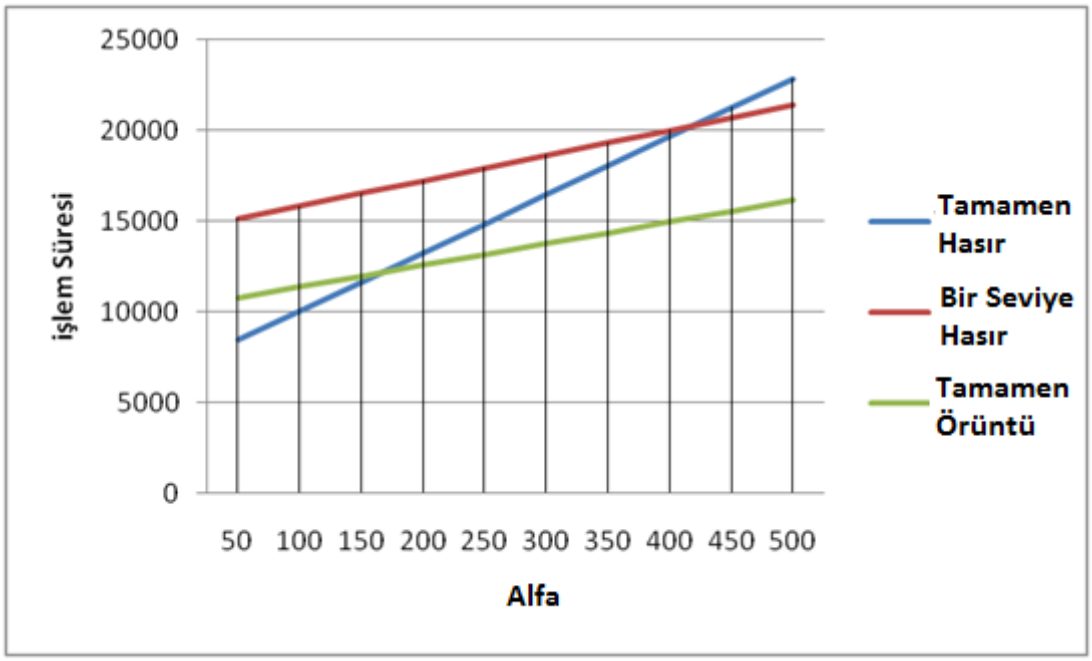
- Mesaj yoğunluğu %100'e yakın olduğu durumlar:
  - Alfa değeri 500 ms'den az olduğu durumlarda, sadece hasır yönteminin daha hızlı sonuçlar verdiği görülmektedir.
  - Mesaj boyunun 25 bayttan az olduğu durumlarda, tamamen örüntünün kullanıldığı yöntem hep daha iyi sonuçlar vermektedir.
  - Mesaj boyunun 100 bayttan büyük olduğu durumlarda, alfa değeri 1000 ms'den küçükse sadece hasır yönteminin kullanıldığı yöntem hep daha hızlı sonuçlar vermektedir.

Şekilleri ayrıntılı inceledikten sonra örnek iki grafik incelenecek. Bu grafiklerin yorumları diğer grafiklerin yorumları için bir örnek teşkil edecektir. Diğer grafiklerde bu örneklerdeki gibi yorumlanabilir.



Şekil 6.3 Mesaj yoğunluğu %50 ve alfa değeri 500 milisaniye olduğu durum için mesaj boyunun ve seçilen yöntemin işlem süresine olan etkisi

Şekil 6.3'de mesaj yoğunluğu %50 ve alfa değeri 500 ms iken mesaj boyu 20 bayttan 200 bayta kadar düzgün bir şekilde arttırıldığı durumda, veri iletim süreleri ms olarak verilmiştir. Grafikte görüldüğü gibi alfa değeri ve mesaj yoğunluğu bu durumdayken mesaj boyu 60 bayta gelene kadar tamamen örüntünün çalıştığı algoritma en kısa sürede veri iletimini sağlarken bir seviye örüntü bir seviye hasır yapısının çalıştığı yöntem, ikinci en iyi yöntem olmuştur. Mesaj boyu bu değere gelene kadar ise tamamen hasır yönteminin çalıştığı yöntem en kötü sonuçları vermektedir. Mesaj boyu 60 bayt ile 150 bayt değerleri arasında iken tamamen hasır yöntemi bir seviye hasır uygulanan yöntemden daha iyi olmaya başlamıştır. 150 bayt değerinden sonra ise tamamen hasır yöntemi tamamen örüntü yönteminden daha iyi sonuçlar vermeye başlamıştır. Sistemin işlemesi sırasında alfa değeri ve mesaj yoğunluğu 500 ve %50 olduğu durumlarda mesaj boyuna göre hangi yöntemin uygulanacağı bu grafiğe göre kararlaştırılabilir.



Şekil 6.4 Mesaj yoğunluğu %100 ve mesaj boyunun 25 bayt olduğu durum için alfanın ve seçilen yöntemin işlem süresine olan etkisi

Şekil 6.4'de mesaj yoğunluğu %100 ve mesaj boyu 25 bayttır. Bu değerler sabit tutularak alfanın 50 ms'den 500 ms'ye kadar değerleri için işlem süreleri gösterilmiştir. Burada görüleceği gibi alfa değerinin küçük olduğu yerlerde tamamen hasır yapısının kullanıldığı yöntem, diğer iki yöntemden daha iyi çalışmaktadır. Tamamen hasır yapısında daha fazla işlem adımı olduğu için alfa değeri büyüdükçe, bu yöntemde işlem süresi çok daha hızlı büyümektedir ve alfanın 150 ms'den büyük değerleri için örüntünün kullanıldığı yöntem daha iyi sonuçlar vermektedir.

6.5. Özel Durumların İncelenmesi

Bu bölümde farklı test durumları için hangi yöntemin kullanılması gerektiğine şekil 6.1 üzerinden bakılacaktır. Örneğin, alfa değeri 40 ms, mesaj yoğunluğu %25 ve mesaj boyu 130 bayt olsun, bu durum tabloda A harfi ile belirtilmiş konumu ifade etmektedir. A harfinin bulunduğu konum için sadece örüntünün kullanıldığı yöntem yaklaşık 15000 ms'de, iki yöntemin karışık kullanıldığı yöntem yaklaşık 20000 ms'de ve tamamen hasır yönteminin kullanıldığı durumda ise yaklaşık 10000 ms'de veri iletişimi gerçekleşmektedir. Bu durumda tamamen hasır yöntemini kullanmak daha mantıklı olacaktır.

Alfanın 100 ms, mesaj yoğunluğunun %25 ve mesaj boyunun 50 bayt olduğu durumda (B noktası) grafikten de görüleceği gibi sadece örüntünün kullanıldığı yöntem en az işlem masrafına sahip yöntemdir.

Alfanın 100 ms, mesaj yoğunluğunun %25 ve mesaj boyunun 150 bayt olduğu durum C noktası olarak gösterilmiştir. Bu noktada, B noktasından farklı olarak mesaj boyu arttığı ve bundan dolayı örüntüdeki dar boğazlar etkisini artıracağı için hasır yöntemine doğru geçiş daha verimli olacaktır.

Alfa değeri 100 ms, mesaj yoğunluğu %75 olduğu durumlarda D noktasının bulunduğu grafiğe bakıldığında görüleceği gibi hasır yöntemi her durum için daha mantıklıdır.

Alfa değerinin yüksek olduğu E noktasında, beklendiği gibi sadece hasır yapısının kullanıldığı yöntem işlem zamanı olarak daha az vakit almaktadır.

Alfa değerinin küçük ve işlem yoğunluğunun yüksek oranda olduğu F noktasında ise grafikten de anlaşılacağı gibi tamamen hasır yöntemini kullanmak daha verimli olmaktadır.

Bunlar gibi, istenilen alfa, mesaj yoğunluğu ve mesaj boyu değerleri için, verilen değerler gafiklerde yerine konarak, hangi yöntemin daha iyi çalışacağı tespit edilebilir.

7. SONUÇ

RTI'ın yönettiği nesne yönetimi servisinin modellendiği bu çalışmada, öne sürülen örüntünün hasır yöntemi ile birlikte gerçekleştirimi ve bu iki yöntemin karşılaştırılması gösterilmiş oldu. Aynı zamanda 2 boyutlu torus yapılarında fotonik ağlarla veri iletişimini sağlayabilecek bir tasarım gerçekleştirildi. Fotonik ağlar manyetik halkalarla yönlendirilerek ve iletişim örüntüleri kullanılarak rotalama problemine bu tür uygulamalar için etkin bir çözüm bulunmuş olundu.

Veri iletişimi için gerçekleştirilen yöntem aslında veri dağıtımını yöneten bir yazılım çözümüdür. Torus yapısındaki ağın fotonik ağlarla tasarlanması ise bir donanımsal çözümdür. Tez çalışmasında, bu iki yöntem birleştirilerek, RTI'ın dizaynında hem yazılımsal hem de donanımsal (hardware-software codesign) bir çözüm ortaya konulmuştur.

Bunlarla birlikte, veri iletişimi için önerilen örüntünün olumlu ve olumsuz tarafları incelendi. Örüntüde, verilerin iletimi için işlem sayısı oldukça az olması ciddi bir fayda sağlarken; sistemdeki verinin artması veya bir paketteki veri miktarının artması sonucu oluşan dar boğazlar bir olumsuzluk oluşturmaktadır. Bu olumsuzluklar hasır yöntemine geçişlerle azaltılmaya çalışıldı. Sistem gereksinimleri göz önünde bulundurularak hangi yöntemin kullanılmasının daha mantıklı olacağı incelendi.

Bu çalışmada RTI'ın servislerinden deklarasyon yönetimi servisi ile nesne yönetimi servisi için çözüm yaklaşımı önerilmiştir. Bununla birlikte nesne yönetimi servisi önerilen yöntemlerle gerçekleştirilmiş ve bu servisin analizi yapılmıştır. İleriki çalışmalarda diğer servisler için de çözümler gerçekleştirilerek tam bir RTI çözümü oluşturulması hedeflenmelidir.

KAYNAKLAR

[1] Brown, Martin. "Comparing Traditional Grids With High – Performance Computing." IBM. Jun 13, **2006**.

[2] "Parallel Processing." *Search Data Center*. March 27, 2007. Retrieved March 29, **2008**.

[3] "Programming Models." Tomesani. Retrieved March 29, **2008**.

[4] Biberman, A., Preston, K., Hendry, G., Sherwood-Droz, N., Chan, J., Levy, J. S., Lipson, M., Bergman, K. 2011. Photonic Network-On-Chip Architectures Using Multilayer Deposited Siliconmaterials For High–Performance Chip Multiprocessors. *ACM Journal on Emerging Technologies in Computing Systems*, Vol. 7, Issue 2, July **2011**, DOI=10.1145/1970406.1970409.

[5] Chan, J.,Hendry, G., Biberman, A., Bergman, K. 2010. Architectural Exploration of Chip-Scale Photonic Interconnection Network Designs Using Physical-Layer Analysis," *Journal of Lightwave Technology*, vol.28, no.9, pp.1305-1315, May, **2010** DOI=10.1109/JLT.2010.2044231.

[6] Debaes, C.,Artundo, I., Heirman, W., Loperena, M., Van Campenhout, J., Thienpont, H. 2009. Architecturalstudy of reconfigurable Photonic Networks-on-Chip For Multi-Core Processors. LEOS Annual Meeting Conference Proceedings, 2009. LEOS '09. IEEE , 4-8 Oct. **2009**.

[7] Kayhan İmre, Nevzat Sevim, "The High Level Architecture (HLA) on Photonic Torus: Hardware and Software Co-design" *8 th EUROSIM Congress*, Cardiff, Wales, United Kingdom, 10-13 September **2013.**

[8] Intel® Xeon® Phi™ coprocessors http://www.intel.com/xeonphi

[9] Shah, A. 2011. Adapteva Aims 64-core Chip at Tablets, Smartphones. *PCWorld Magazine*, October 3, **2011**.

[10] Pan, K., Turner, S.J., Cai, W., Li, Z. 2011. A dynamic sort-based DDM matching algorithm for HLA applications. *ACM Trans. Model. Comput. Simul.* 21, 3, Article 17 (February **2011**), 17 pages. DOI=10.1145/1921598.1921601 http://doi.acm.org/10.1145/1921598.1921601

[11] Perumalla, K.S. **2006**. Parallel and distributed simulation: traditional techniques and recent advances. In Proceedings of the 38th conference on Winter simulation (WSC '06), L. Felipe Perrone, Barry G. Lawson, Jason Liu, and Frederick P. Wieland (Eds.). Winter Simulation Conference 84-95.

[12] Strassburger, S., Schulze, T., Fujimoto R. **2008**. Future trends in distributed simulation and distributed virtual environments: results of a peer study. In Proceedings of the 40th Conference on Winter Simulation (WSC '08), Scott Mason, Ray Hill, Lars Moench, and Oliver Rose (Eds.). Winter Simulation Conference 777-785.

[13] IEEE **2010**. IEEE STD 1516.1-2010 Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification.

[14] Imre, K. M., Baransel C., And Artuner, H. 2011. Efficient and Scalable Routing Algorithms for Collective Communication Operations on 2D All-Port Torus Networks*. International Journal of Parallel Programming, Springer*, DOI : 10.1007/s10766-011-0169-2, **2011**.

[15] Carr, C. E., et al. "Laminar organization of the afferent and efferent systems of the torus semicircularis of gymnotiform fish: morphological substrates for parallel processing in the electrosensory system." Journal of Comparative Neurology 203.4 (**1981**): 649-670.

[16] Baransel, C., Imre, K., Artuner, H. 2010. New Parallel Matrix Multiplication Algorithms for Wormhole–Routed All–Port 2D/3D

Torus Networks. Mathematical Methods in Engineering International Symposium (MME '10), Coimbra, Portugal, ISBN 978-989-8331-11-3, 21–24 October, **2010**.

[17] Benini, Luca, and Giovanni De Micheli. "Networks on chips: a new SoC paradigm." Computer 35.1 (**2002**): 70-78.

[18] *RTI 1.3-Next Generation Programmer's Guide Version 4*. U.S. Department of Defense **2001**.

[19] Bruck, Jehoshua, et al. "Efficient algorithms for all-to-all communications in multiport message-passing systems." Parallel and Distributed Systems, *IEEE Transactions* on 8.11 (**1997**): 1143-1156.

[20] Lam, Chi Chung, C-H. Huang, and P. Sadayappan. "Optimal algorithms for all-to-all personalized communication on rings and two dimensional tori." *Journal of Parallel and Distributed Computing*43.1 (**1997**): 3-13.

[21] Gustafson, John L. "Reevaluating Amdahl's law." *Communications of the ACM* 31.5 (**1988**): 532-533.

[22] Baer, Jean-Loup (**2010**). Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors. New York: Cambridge University Press. p. 10. ISBN 978-0-521-76992-1.

[23] *RTI 1.3-Next Generation Programmer's Guide Version 5*. U.S. Department of Defense **2002**.

[24] SATO, Ken-ichi, et al. GMPLS-based photonic multilayer router (Hikari router) architecture: an overview of traffic engineering and signaling technology. *Communications Magazine, IEEE*, **2002**, 40.3: 96-101.

[25] Intel® Xeon Phi™ Coprocessor 5110P (60 core) http://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core

EK 1

Bu ek, RTI'ın nesne yönetimi servisi için tezde bahs edilen örüntünün ve hasır yapısının gerçekleştirildiği koddur.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#include <sys/time.h>
#include <sys/types.h>
#include "dataStructers.h" // EK 2 Olarak Verildi

struct Processor mainBoard[MaxN][MaxM];
int ProcNo, MolNo, mashVal, N, M, Time=0, cost=0, alpha;
int molNumMin, molNumMax, molNo;
// [][0]->islem [][1]->derece
int PublishPatern[MaxN][2], lengthPub;
// islemler icin 1->Gather, 2-> Multicast 3->Exchange -1->Mash
int SubscribePatern[MaxN][2], lengthSub;
int costs[MaxN][MaxM][4][3];

FILE *subs;
FILE *pubs;
int readPublish();
int readSubscribe();

int yollar[16][4]={ {0,0,0,0}, {0,1,0,0}, {2,0,0,0}, {1,0,0,0},
                    {1,1,1,1}, {0,2,1,1}, {3,0,1,1}, {1,2,1,1},
                    {2,2,2,2}, {3,1,2,2}, {0,3,2,2}, {2,1,2,2},
                    {3,3,3,3}, {1,3,3,3}, {2,3,3,3}, {3,2,3,3} };

int yollar2[16][4]={{0,0,0,0}, {0,0,1,1}, {0,0,2,2}, {0,0,3,3},
                 {1,1,0,0}, {1,1,1,1}, {1,1,2,2}, {1,1,3,3},
                 {2,2,0,0}, {2,2,1,1}, {2,2,2,2}, {2,2,3,3},
                 {3,3,0,0}, {3,3,1,1}, {3,3,2,2}, {3,3,3,3} };

int yonler[5][2]={{-1,0},{0,-1},{0,1},{1,0},{0,0}};

void GenerateMolecules(struct Processor mainBoard[][MaxM],int N, int M, int
molNumMin, int molNumMax, int *MolNo)
{
   int i, j, k;
   srand(time(0));

   for(i=0;i<N;i++)
      for(j=0;j<M;j++)
      {
         mainBoard[i][j].numMol=molNumMin+rand()%(molNumMax-molNumMin+1);

         for(k=0 ; k<mainBoard[i][j].numMol ; k++)
         {
            mainBoard[i][j].molecules[k].no=(*MolNo)++;
            mainBoard[i][j].molecules[k].X=rand()%1000000;
            mainBoard[i][j].molecules[k].Y=rand()%1000000;
         }
      }
}
```

```
void setPublishPatern(int depth, int mashVal)
{
   if(us4(depth-1)==N) return;
   if(depth==mashVal) //Mash
   {
      PublishPatern[lengthPub][0]=-1;
      PublishPatern[lengthPub++][1]=depth;
      return;
   }

   PublishPatern[lengthPub][0]=1; // Gather
   PublishPatern[lengthPub++][1]=depth;

   setPublishPatern(depth+1,mashVal);

   PublishPatern[lengthPub][0]=3;   // Exchange
   PublishPatern[lengthPub++][1]=depth;

   PublishPatern[lengthPub][0]=2;   // Multicast
   PublishPatern[lengthPub++][1]=depth;

}

void setSubscribePatern(int depth, int mashVal)
{
   if(us4(depth-1)==N) return;
   if(depth==mashVal) // Mash
   {
      SubscribePatern[lengthSub][0]=-1;
      SubscribePatern[lengthSub++][1]=depth;
      return;
   }

   SubscribePatern[lengthSub][0]=1; // Gather
   SubscribePatern[lengthSub++][1]=depth;

   SubscribePatern[lengthSub][0]=3; // Exchange
   SubscribePatern[lengthSub++][1]=depth;

   setSubscribePatern(depth+1,mashVal);

   SubscribePatern[lengthSub][0]=2; // Multicast
   SubscribePatern[lengthSub++][1]=depth;

}

int kont2(int a, int b, int sira, int tx1, int ty1, int tx2, int ty2, int x, int y)
{
   int i, cnt=mainBoard[a][b].numRout;

   for(i=0;i<cnt;i++)
      if(mainBoard[a][b].routingTable[i][0]==sira)
         if(mainBoard[a][b].routingTable[i][1]==tx1)
            if(mainBoard[a][b].routingTable[i][2]==ty1)
               if(mainBoard[a][b].routingTable[i][3]==tx2)
                  if(mainBoard[a][b].routingTable[i][4]==ty2)
                     if(mainBoard[a][b].routingTable[i][5]==x)
                        if(mainBoard[a][b].routingTable[i][6]==y)
                           return 0;
   return 1;
```

```
}

void paketGonder(int a, int b, int c, int d, int v)
{
   int i;
   for(i=0;i<4;i++)
      if(costs[a][b][i][0]==0)
      {
         costs[a][b][i][0]=v;
         costs[a][b][i][1]=c;
         costs[a][b][i][2]=d;
         return;
      }
      else if(costs[a][b][i][1]==c && costs[a][b][i][2]==d)
      {
         costs[a][b][i][0]+=v;
         return;
      }

}

int subscribeGather(int depth, int sira)
{
   int i, j, x=-1, y=-1, k, t;
   int I=N/us4(depth), J=M/us4(depth), T=us4(depth-1);

   for(i=0;i<I;i++)
      for(j=0;j<J;j++)
      {
         x=i*us4(depth); y=j*us4(depth);
         for(t=0;t<T;t++)
         {
            for(k=0;k<16;k++)
            {
               int sx=x+yollar[k][0]*T, sy=y+yollar[k][1]*T;
               int tx=x+yollar[k][2]*T, ty=y+yollar[k][3]*T;
               struct SubscribePack *p, *pt;

               for(p=mainBoard[sx][sy].subscribes ; p ; p=pt,
mainBoard[sx][sy].subscribes=pt )
               {
                  pt=p->n;
                  if(kont2(tx,ty,sira,p->targetX1,p->targetY1,p->targetX2,p-
>targetY2,sx,sy))
                  {
                     int cnt=mainBoard[tx][ty].numRout;
                     mainBoard[tx][ty].routingTable[cnt][0]=sira;
                     mainBoard[tx][ty].routingTable[cnt][1]=p->targetX1;
                     mainBoard[tx][ty].routingTable[cnt][2]=p->targetY1;
                     mainBoard[tx][ty].routingTable[cnt][3]=p->targetX2;
                     mainBoard[tx][ty].routingTable[cnt][4]=p->targetY2;
                     mainBoard[tx][ty].routingTable[cnt][5]=sx;
                     mainBoard[tx][ty].routingTable[cnt][6]=sy;
                     mainBoard[tx][ty].numRout++;
                     p->n=mainBoard[tx][ty].substemp;
                     mainBoard[tx][ty].substemp=p;
                     paketGonder(sx,sy,tx,ty,p->volume);
                  }
               }
            }
            for(k=0;k<4;k++)
```

```
          {
            int sx=x+k*T, sy=y+k*T;
            mainBoard[sx][sy].subscribes = mainBoard[sx][sy].substemp;
            mainBoard[sx][sy].substemp=NULL;
          }
          x++, y++;
        }
      }
   return 0;
}


int subscribeMulticast(int depth, int sira)
{
   int i, j, x=-1, y=-1, k, t, l;
   int I=N/us4(depth), J=M/us4(depth), T=us4(depth-1);

   for(i=0;i<I;i++)
     for(j=0;j<J;j++)
     {
       x=i*us4(depth); y=j*us4(depth);
       for(t=0;t<T;t++)
       {
         for(k=0;k<16;k++)
         {
           int sx=x+yollar[k][2]*T, sy=y+yollar[k][3]*T;
           int tx=x+yollar[k][0]*T, ty=y+yollar[k][1]*T;
           struct SubscribePack *p, *pt;
           for(p=mainBoard[sx][sy].subscribes ; p ; p=p->n )
           {
             if(kont2(tx,ty,sira,p->targetX1,p->targetY1,p->targetX2,p-
>targetY2,sx,sy))
             {
               pt=getSubscribePack();  *pt=*p;
               int cnt=mainBoard[tx][ty].numRout;
               mainBoard[tx][ty].routingTable[cnt][0]=sira;
               mainBoard[tx][ty].routingTable[cnt][1]=p->targetX1;
               mainBoard[tx][ty].routingTable[cnt][2]=p->targetY1;
               mainBoard[tx][ty].routingTable[cnt][3]=p->targetX2;
               mainBoard[tx][ty].routingTable[cnt][4]=p->targetY2;
               mainBoard[tx][ty].routingTable[cnt][5]=sx;
               mainBoard[tx][ty].routingTable[cnt][6]=sy;
               mainBoard[tx][ty].numRout++;
               pt->n=mainBoard[tx][ty].substemp;
               mainBoard[tx][ty].substemp=pt;
               paketGonder(sx,sy,tx,ty,p->volume);
             }
           }
         }
         for(k=0;k<4;k++)
           for(l=0;l<4;l++)
           {
             int sx=x+k*T, sy=y+l*T;
             struct SubscribePack *p, *pt;
             if(sira==0)
             {
               for(p=mainBoard[sx][sy].subscribes ; p ; p=pt){ pt=p->n; free(p); }
               for(p=mainBoard[sx][sy].substemp   ; p ; p=pt){ pt=p->n; free(p); }
               mainBoard[sx][sy].subscribes=mainBoard[sx][sy].substemp=NULL;
             }
             else
```

```
            {
              for(p=mainBoard[sx][sy].subscribes ; p ; p=pt){ pt=p->n; free(p); }
              mainBoard[sx][sy].subscribes=mainBoard[sx][sy].substemp;
              mainBoard[sx][sy].substemp=NULL;
            }
          }
        x++, y++;
      }
    }
  return 0;
}


int subscribeExchange(int depth, int sira)
{

  int i, j, x=-1, y=-1, k, t;
  int I=N/us4(depth), J=M/us4(depth), T=us4(depth-1);

  for(i=0;i<I;i++)
    for(j=0;j<J;j++)
    {
      x=i*us4(depth); y=j*us4(depth);
      for(t=0;t<T;t++)
      {
        for(k=0;k<16;k++)
        {
          int sx=x+yollar2[k][0]*T, sy=y+yollar2[k][1]*T;
          int tx=x+yollar2[k][2]*T, ty=y+yollar2[k][3]*T;
          struct SubscribePack *p, *pt;

          for(p=mainBoard[sx][sy].subscribes ; p ; p=p->n)
          {

            if(kont2(tx,ty,sira,p->targetX1,p->targetY1,p->targetX2,p-
>targetY2,sx,sy))
              {
                pt=getSubscribePack(); *pt=*p;
                int cnt=mainBoard[tx][ty].numRout;
                mainBoard[tx][ty].routingTable[cnt][0]=sira;
                mainBoard[tx][ty].routingTable[cnt][1]=p->targetX1;
                mainBoard[tx][ty].routingTable[cnt][2]=p->targetY1;
                mainBoard[tx][ty].routingTable[cnt][3]=p->targetX2;
                mainBoard[tx][ty].routingTable[cnt][4]=p->targetY2;
                mainBoard[tx][ty].routingTable[cnt][5]=sx;
                mainBoard[tx][ty].routingTable[cnt][6]=sy;
                mainBoard[tx][ty].numRout++;
                pt->n=mainBoard[tx][ty].substemp;
                mainBoard[tx][ty].substemp=pt;
                paketGonder(sx,sy,tx,ty,p->volume);
              }
            }
          }

          for(k=0;k<4;k++)
          {
            int sx=x+k*T, sy=y+k*T;
            struct SubscribePack *p, *pt;
            for(p=mainBoard[sx][sy].subscribes ; p ; p=pt){ pt=p->n; free(p); }
            mainBoard[sx][sy].subscribes=mainBoard[sx][sy].substemp;
            mainBoard[sx][sy].substemp=NULL;
```

```
          }
          x++, y++;
        }
      }
   return 0;

}

int kont(int a, int b, int x1, int y1, int x2, int y2)
{
   int mesafe(int a, int b, int x, int y)
   {
     return ( Min( ABS(a-x), Min(ABS(a-x-N), ABS(a-x+N))  ) )+
          ( Min( ABS(b-y), Min(ABS(b-y-M), ABS(b-y+M))  ) );
   }

   if( mesafe(x1,y1,x2,y2)>mesafe(x1,y1,a,b) ) return 1;
   return 0;
}

int subscribeMash(int depth, int sira)
{
   int i, x, y, j, k, K, F, t, T=us4(depth-1), time=0;
   int minT, minX, minY, minK;
   struct SubscribePack *p, *pt;


   for(i=0;i<T;i++)
   {
     for(x=i;x<N;x+=T)
          for(y=i;y<M;y+=T)
          {
            for(p=mainBoard[x][y].subscribes ; p ; p=p->n)
            {
              int cnt=mainBoard[x][y].numRout;
              mainBoard[x][y].routingTable[cnt][0]=sira;
              mainBoard[x][y].routingTable[cnt][1]=p->targetX1;
              mainBoard[x][y].routingTable[cnt][2]=p->targetY1;
              mainBoard[x][y].routingTable[cnt][3]=p->targetX2;
              mainBoard[x][y].routingTable[cnt][4]=p->targetY2;
              mainBoard[x][y].routingTable[cnt][5]=-1;
              mainBoard[x][y].routingTable[cnt][6]=-1;
              mainBoard[x][y].numRout++;
            }
          }
     for(t=1;t>=0;t--)
     {
       for(x=i;x<N;x+=T)
          for(y=i;y<M;y+=T)
          {
            for(p=mainBoard[x][y].subscribes ; p ; p=p->n)
              for(K=0,k=t; K<2 ; K++, k=3-k)
              {
                //printf("-------1----------\n");

                pt=getSubscribePack();  *pt=*p; pt->count=0; pt->n=NULL;
                if(mainBoard[x][y].mashSubQ[k]==NULL)
                {
                  pt->bitis=time+pt->volume;
                  mainBoard[x][y].mashSSon[k]=mainBoard[x][y].mashSubQ[k]=pt;
```

```
                }
                else
                {
                  pt->bitis=mainBoard[x][y].mashSSon[k]->bitis+pt->volume;
                  mainBoard[x][y].mashSSon[k]->n=pt;
                  mainBoard[x][y].mashSSon[k]=pt;
                }
              }
          }
        F=1;
        while(F)
        {
          F=0; minT=INT_MAX;
          for(x=i;x<N;x+=T)
            for(y=i;y<M;y+=T)
            {
              for(k=t,K=0; K<2 ; K++, k=3-k)
              {
                if(mainBoard[x][y].mashSubQ[k] && mainBoard[x][y].mashSubQ[k]-
>bitis<minT)
                {
                  F=1;
                  minT=mainBoard[x][y].mashSubQ[k]->bitis;
                  minX=x; minY=y; minK=k;
                }
              }
            }
          if(!F) break;
          p=mainBoard[minX][minY].mashSubQ[minK];
          p->count++;
          mainBoard[minX][minY].mashSubQ[minK]=p->n;
          int tx=minX+yonler[minK][0]*T, ty=minY+yonler[minK][1]*T;
          tx=(tx<0)? tx+N : (tx>=N)?tx%N:tx;
          ty=(ty<0)? ty+M : (ty>=M)?ty%M:ty;
          time=p->bitis;
          if(kont2(tx,ty,sira,p->targetX1,p->targetY1,p->targetX2,p-
>targetY2,minX,minY))
          {
            //printf("-------2-------  %d %d %d\n",p->sourceX,p->sourceY,p->count);
            pt=getSubscribePack(); *pt=*p; pt->n=NULL;
            int cnt=mainBoard[tx][ty].numRout;
            mainBoard[tx][ty].routingTable[cnt][0]=sira;
            mainBoard[tx][ty].routingTable[cnt][1]=p->targetX1;
            mainBoard[tx][ty].routingTable[cnt][2]=p->targetY1;
            mainBoard[tx][ty].routingTable[cnt][3]=p->targetX2;
            mainBoard[tx][ty].routingTable[cnt][4]=p->targetY2;
            mainBoard[tx][ty].routingTable[cnt][5]=3-minK;
//          mainBoard[tx][ty].routingTable[cnt][6]=minY;
            mainBoard[tx][ty].numRout++;
            if( ( (minK==1 || minK==0) && pt->count<(N/T)/2) || ( pt->count <
(N/T)/2-1 ) )
            {
//            printf("-------3-------  %d %d %d\n",p->sourceX,p->sourceY,p->count);
              if(mainBoard[tx][ty].mashSubQ[minK]==NULL)
              {
                pt->bitis=time+pt->volume;

  mainBoard[tx][ty].mashSSon[minK]=mainBoard[tx][ty].mashSubQ[minK]=pt;
              }
              else
              {
```

```
                pt->bitis=mainBoard[tx][ty].mashSSon[minK]->bitis+pt->volume;
                mainBoard[tx][ty].mashSSon[minK]->n=pt;
                mainBoard[tx][ty].mashSSon[minK]=pt;
              }
            }
            else free(pt);}
            p->n=mainBoard[tx][ty].subscribes;
            mainBoard[tx][ty].subscribes=p;
          }
          else free(p);
        }
      }

      if(lengthSub==1) ///// Paketleri dusur
      {
        for(x=i;x<N;x+=T)
          for(y=i;y<M;y+=T)
          {
            for(p=mainBoard[x][y].subscribes ; p ; p=pt)
            {
              pt=p->n; free(p);
            }
            mainBoard[x][y].subscribes=NULL;
          }
      }
    }
    return time;
}

int kesisiyormu(int a, int b, int c, int d, int x, int y, int z, int k)
{
    int ar[2][4];
    if(a>x){ a=a+x; x=a-x; a=a-x;
          b=b+y; y=b-y; b=b-y;
          c=c+z; z=c-z; c=c-z;
          d=d+k; k=d-k; d=d-k; }

    if(x>c || k<b || y>d) return 0;
    return 1;

}

int publishMash(int depth, int sira)
{
    int i, x, y, j, k, K, F, t, T=us4(depth-1), time=0;
    int minT, minX, minY, minK;
    struct UpdatePack *p, *pt;

    for(i=0;i<T;i++)
    {
      for(x=i;x<N;x+=T)
        for(y=i;y<M;y+=T)
        {
          while( p=mainBoard[x][y].updates )
          {
            p->used=(int *)calloc(N*M,sizeof(int));
            *(p->used+(x*M+y))=1;

            for(k=-1;k<4;k++)
            {
              for(t=0;t<mainBoard[x][y].numRout;t++)
```

```
                if(mainBoard[x][y].routingTable[t][0]==sira &&
mainBoard[x][y].routingTable[t][5]==k &&
kesisiyormu(mainBoard[x][y].routingTable[t][1], mainBoard[x][y].routingTable[t][2],
                            mainBoard[x][y].routingTable[t][3],
mainBoard[x][y].routingTable[t][4],
                            p->targetX1, p->targetY1, p->targetX2, p->targetY2)
                  )
                {
                  pt=getUpdatePack();   *pt=*p; pt->n=NULL;

                  if(k==-1)
                  {
                    pt->n=mainBoard[x][y].updtemp;
                    mainBoard[x][y].updtemp=pt;
                  }
                  else if(mainBoard[x][y].mashPubQ[k]==NULL)
                  {
                    pt->bitis=time+pt->volume;
                    mainBoard[x][y].mashPSon[k]=mainBoard[x][y].mashPubQ[k]=pt;
                  }
                  else
                  {
                    pt->bitis=mainBoard[x][y].mashPSon[k]->bitis+pt->volume;
                    mainBoard[x][y].mashPSon[k]->n=pt;
                    mainBoard[x][y].mashPSon[k]=pt;
                  }
                  t=mainBoard[x][y].numRout;
                }
            }
            mainBoard[x][y].updates=mainBoard[x][y].updates->n;
            free(p);
          }
        }

     F=1;
     while(F)
     {
       F=0; minT=INT_MAX;
       for(x=i;x<N;x+=T)  ////////// En kucuk bitis olani bul
          for(y=i;y<M;y+=T)
          {
            for(k=0;k<4;k++)
              if(mainBoard[x][y].mashPubQ[k] && mainBoard[x][y].mashPubQ[k]-
>bitis<minT)
                {
                  F=1;
                  minT=mainBoard[x][y].mashPubQ[k]->bitis;
                  minX=x; minY=y; minK=k;

                }
          }

       if(!F) break;
       p=mainBoard[minX][minY].mashPubQ[minK];
       mainBoard[minX][minY].mashPubQ[minK]=p->n; //// paketi listeden cikar
       if(mainBoard[minX][minY].mashPubQ[minK]==NULL)
mainBoard[minX][minY].mashPSon[minK]=NULL;
       int tx=minX+yonler[minK][0]*T, ty=minY+yonler[minK][1]*T; ////// Paketin
gidecegi islemciyi bul
       tx=(tx<0)? tx+N : (tx>=N)?tx%N:tx;
       ty=(ty<0)? ty+M : (ty>=M)?ty%M:ty;
```

```
       //printf("--> %d %d %d %d %d %d\n",minX,minY,minK,p->sourceX, p->sourceY, p-
>bitis);

       if( *(p->used+(tx*M+ty))==1 ) { free(p); continue; }
       *(p->used+(tx*M+ty))=1;
       time=p->bitis;

       for(k=-1;k<4;k++) //////// gidilecek islemcide hangi listelere eklenecegini
hesapla
       if(k!=3-minK)
       {

          for(t=0;t<mainBoard[tx][ty].numRout;t++)
            if(mainBoard[tx][ty].routingTable[t][0]==sira &&
mainBoard[tx][ty].routingTable[t][5]==k &&
kesisiyormu(mainBoard[tx][ty].routingTable[t][1],
mainBoard[tx][ty].routingTable[t][2],
                     mainBoard[tx][ty].routingTable[t][3],
mainBoard[tx][ty].routingTable[t][4],
                     p->targetX1, p->targetY1, p->targetX2, p->targetY2)
            )
            {
              pt=getUpdatePack();   *pt=*p; pt->n=NULL;

              if(k==-1)
              {
//              printf("------->%d %d %d %d %d\n",tx,ty,k,pt->sourceX,pt-
>sourceY,pt->bitis);
                pt->n=mainBoard[tx][ty].updtemp;
                mainBoard[tx][ty].updtemp=pt;
              }
              else if(mainBoard[tx][ty].mashPubQ[k]==NULL)
              {
                pt->bitis=time+pt->volume;
                mainBoard[tx][ty].mashPSon[k]=mainBoard[tx][ty].mashPubQ[k]=pt;
              }
              else
              {
                pt->bitis=mainBoard[tx][ty].mashPSon[k]->bitis+pt->volume;
                mainBoard[tx][ty].mashPSon[k]->n=pt;
                mainBoard[tx][ty].mashPSon[k]=pt;
              }
              t=mainBoard[tx][ty].numRout;
            }
       }
       free(p);
     }

     for(x=i;x<N;x+=T)
          for(y=i;y<M;y+=T)
          {
            if(lengthPub==1)
              for(p=mainBoard[x][y].updtemp ; p ; p=p->n)
              {
//              printf("(%d,%d)'den (%d,%d)'ye update ulasti\n",p->sourceX, p-
>sourceY, x, y);
                pt=p->n; free(p);
                mainBoard[x][y].updtemp=mainBoard[x][y].updates=NULL;
              }
            else
            {
```

```
                mainBoard[x][y].updates=mainBoard[x][y].updtemp;
                mainBoard[x][y].updtemp=NULL;
              }
            }

   }
   return time;
}

int kont3(int a, int b, int k, struct UpdatePack *p)
{
   int i;
   for(i=0;i<k;i++)
     if(  mainBoard[a][b].routingTable[i][0]==mainBoard[a][b].routingTable[k][0] &&
        mainBoard[a][b].routingTable[i][5]==mainBoard[a][b].routingTable[k][5] &&
        mainBoard[a][b].routingTable[i][6]==mainBoard[a][b].routingTable[k][6] &&
        kesisiyormu(mainBoard[a][b].routingTable[i][1],
mainBoard[a][b].routingTable[i][2],
                mainBoard[a][b].routingTable[i][3],
mainBoard[a][b].routingTable[i][4],
                p->targetX1, p->targetY1, p->targetX2, p->targetY2)
       )
        return 0;

   return 1;
}

void publishGo(int sira)
{
   int i, j, k;
   struct UpdatePack *p, *pt;

   for(i=0;i<N;i++)
     for(j=0;j<M;j++)
     {
        for(k=0;k<mainBoard[i][j].numRout;k++)
          if( mainBoard[i][j].routingTable[k][0]==sira )
          {

             for(p=mainBoard[i][j].updates ; p ; p=p->n)
               if(  kesisiyormu(mainBoard[i][j].routingTable[k][1],
mainBoard[i][j].routingTable[k][2],
                        mainBoard[i][j].routingTable[k][3],
mainBoard[i][j].routingTable[k][4],
                        p->targetX1, p->targetY1, p->targetX2, p->targetY2)
                  &&kont3(i,j,k,p)   )
               {
                  int tx=mainBoard[i][j].routingTable[k][5],
ty=mainBoard[i][j].routingTable[k][6];
                  pt=getUpdatePack(); *pt=*p;
                  pt->n=mainBoard[tx][ty].updtemp;
                  mainBoard[tx][ty].updtemp=pt;
                  paketGonder(i,j,tx,ty,p->volume);
               }
          }

     }
   for(i=0;i<N;i++)
     for(j=0;j<M;j++)
     {
        if(sira==lengthPub-1)
```

```
      {
        for(p=mainBoard[i][j].updates ; p ; p=pt){ pt=p->n; free(p); }
        for(p=mainBoard[i][j].updtemp ; p ; p=pt)
        {
          pt=p->n; free(p);
        }
        mainBoard[i][j].updates=mainBoard[i][j].updtemp=NULL;
      }
      else
      {
        for(p=mainBoard[i][j].updates ; p ; p=pt){ pt=p->n; free(p); }
        mainBoard[i][j].updates=mainBoard[i][j].updtemp;
        mainBoard[i][j].updtemp=NULL;
      }
    }
}

int publishGather(int sira)        {  publishGo(sira);  return 1;  }
int publishMulticast(int sira)     {  publishGo(sira);  return 1;  }
int publishExchange(int sira)      {  publishGo(sira);  return 1;  }
  return 1;  }

int Cost()
{
  int i, j, k, res=0;

  for(i=0;i<N;i++)
    for(j=0;j<M;j++)
      for(k=0;k<4;k++)
      {
        if(costs[i][j][k][0]!=0) res=Max(res,costs[i][j][k][0]);
        costs[i][j][k][0]=0;
      }
  return res;
}

void executePaterns()
{
  int i=0, j=0, t=0;

  while(i<lengthSub)
  {

    while(SubscribePatern[i][0]!=PublishPatern[j][0]
        || SubscribePatern[i][1]!=PublishPatern[j][1])
        {
          if(PublishPatern[j][0]==1)    { publishGather(j);    t=Cost();
t+=(t==0)?0:alpha;   }
          else if(PublishPatern[j][0]==2)  { publishMulticast(j); t=Cost();
t+=(t==0)?0:alpha; }
          else if(PublishPatern[j][0]==3)  { publishExchange(j);    t=Cost();
t+=(t==0)?0:alpha;   }
          else if(PublishPatern[j][0]==-1)
          {
            t=publishMash(PublishPatern[j][1],j);
            t+= (t==0)? 0 : ((N/us4(PublishPatern[j][1]-1))/2  +
(M/us4(PublishPatern[j][1]-1))/2)*alpha;
          }
          Time++;
          cost+=t;
          j++; j%=lengthPub; if(j==0 && i<lengthSub) readPublish();
```

```
        }

    if(PublishPatern[j][0]==1)
    {
      publishGather(j); j++;
      subscribeGather(SubscribePatern[i][1],lengthSub-i-1); i++;
      t=Cost(); t+=(t==0)?0:alpha;
    }
    else if(PublishPatern[j][0]==2)
    {
      publishMulticast(j); j++;
      subscribeMulticast(SubscribePatern[i][1],lengthSub-i-1); i++;
      t=Cost(); t+=(t==0)?0:alpha;
    }
    else if(PublishPatern[j][0]==3)
    {
      publishExchange(j); j++;
      subscribeExchange(SubscribePatern[i][1],lengthSub-i-1); i++;
      t=Cost(); t+=(t==0)?0:alpha;
    }
    else if(PublishPatern[j][0]==-1)
    {
      t=publishMash(PublishPatern[j][1],j); j++;
      t+=subscribeMash(SubscribePatern[i][1],lengthSub-i-1);i++;
      t+= (t==0)? 0 : ((N/us4(PublishPatern[j-1][1]-1))/2  +
(M/us4(PublishPatern[j-1][1]-1))/2)*alpha;
    }
    else printf("wrong pattern!!!\n");
    Time++;
    cost+=t;
    j%=lengthPub;  if(j==0 && i<lengthSub) readPublish();

  }
}

int readSubscribe()
{
  int time, t, i;
  int sx, sy, tx1, ty1, tx2, ty2;
  struct SubscribePack *p;

  if(fscanf(subs," %d %d",&time,&t)==-1) return 0;
  for(i=0;i<t;i++)
  {
    fscanf(subs," %d %d %d %d %d %d",&sx,&sy,&tx1,&ty1,&tx2,&ty2);
  //subscribes.txt
    p=getSubscribePack();p->volume=subsPackVol;
    p->sourceX=sx;     p->sourceY=sy;
    p->targetX1=tx1;   p->targetY1=ty1;
    p->targetX2=tx2;   p->targetY2=ty2;
    p->n=mainBoard[sx][sy].subscribes;
    mainBoard[sx][sy].subscribes=p;
  }
  return 1;
}

int readPublish(int V)
{
  int i, time, t, sx, sy, tx1, ty1, tx2, ty2, v;
  struct UpdatePack *updP;
```

```
   if(fscanf(pubs," %d %d",&time,&t)==-1) return 0;
   if(time==0 && t==0) return 0;
   v=V;
   for(i=0;i<t;i++)
   {
      fscanf(pubs," %d %d %d %d %d %d",&sx,&sy,&tx1,&ty1,&tx2,&ty2);
      updP = getUpdatePack(); updP->volume=v;
      updP->sourceX=sx;  updP->sourceY=sy;
      updP->targetX1=tx1; updP->targetY1=ty1;
      updP->targetX2=tx2; updP->targetY2=ty2;
      updP->n = mainBoard[sx][sy].updates;
      mainBoard[sx][sy].updates = updP;
   }
   return 1;
}

void executeSubscribe()
{
   int i, t;
   for(i=0;i<lengthSub;i++)
   {
      if(SubscribePatern[i][0]==-1)
      {
         t=subscribeMash(SubscribePatern[i][1],lengthSub-i-1);
         t+= (t==0)? 0 : (((N/us4(SubscribePatern[i][1]-1))/2  +
(M/us4(SubscribePatern[i][1]-1))/2)*alpha) ;
      }
      else if(SubscribePatern[i][0]==1)
      {
         subscribeGather(SubscribePatern[i][1],lengthSub-i-1);
         t=Cost();
         t+=(t==0)?0:alpha;
      }
      else if(SubscribePatern[i][0]==2)
      {
         subscribeMulticast(SubscribePatern[i][1],lengthSub-i-1);
         t=Cost();
         t+=(t==0)?0:alpha;
      }
      else if(SubscribePatern[i][0]==3)
      {
         subscribeExchange(SubscribePatern[i][1],lengthSub-i-1);
         t=Cost();
         t+=(t==0)?0:alpha;
      }
      Time++;
      cost+=t;
   }
}

int main()
{
   int i, j, k, t, MASHVAL, ALPHA, MESAJB;

   ProcNo=MolNo=mashVal=N=M=Time=cost=alpha=0;
   molNumMin=molNumMax=molNo=0;
   for(i=0;i<MaxN;i++) for(j=0;j<2;j++) PublishPatern[i][j]=SubscribePatern[i][j]=0;
   lengthPub=lengthSub=0;
   for(i=0;i<MaxN;i++) for(j=0;j<MaxM;j++) for(k=0;k<4;k++) for(t=0;t<3;t++)
costs[i][j][k][t]=0;
```

```
   for(i=0;i<MaxN;i++)
     for(j=0;j<MaxM;j++)
     {

   mainBoard[i][j].no=mainBoard[i][j].X=mainBoard[i][j].Y=mainBoard[i][j].numMol=main
Board[i][j].numRout=0;
          for(k=0;k<5000;k++) for(t=0;t<7;t++) mainBoard[i][j].routingTable[k][t]=0;
          mainBoard[i][j].updates=mainBoard[i][j].updtemp=NULL;
          mainBoard[i][j].subscribes=mainBoard[i][j].substemp=NULL;
          for(k=0;k<4;k++)
               mainBoard[i][j].mashSubQ[k]=mainBoard[i][j].mashSSon[k]=NULL;
          for(k=0;k<4;k++)
               mainBoard[i][j].mashPubQ[k]=mainBoard[i][j].mashPSon[k]=NULL;
       }


     FILE *f=fopen("initial.txt","r");
     subs=fopen("subscribes.txt","r");
     pubs=fopen("publishes.txt","r");


     fscanf(f," %d %d %d",&N,&M,&ALPHA);      // initial.txt
     fscanf(f," %d %d", &molNumMin, &molNumMax);          // initial.txt
     mashVal=MASHVAL;  alpha=ALPHA;
     /*
      * 16x16 icin
      * MashVal=1 ise  tamamen mesh
      * MashVal=2 ise 1 seviye mesh
      * MashVal=3 ise 0 seviye mesh
      * */

     for(i=0;i<N;i++)
       for(j=0;j<M;j++)
       {
         mainBoard[i][j].X=i;
         mainBoard[i][j].Y=j;
         mainBoard[i][j].no=ProcNo++;
       }

     setPublishPatern(1,mashVal);
     setSubscribePatern(1,mashVal);

     printf("{%d,%d},",PublishPatern[i][0], PublishPatern[i][1]); printf("\n");
     printf("{%d,%d},",SubscribePatern[i][0], SubscribePatern[i][1]);printf("\n");

     readSubscribe();
     executeSubscribe();

     while(1)
     {
       if(readSubscribe()==0 &&readPublish(MESAJB)==0 ) break;
       executePaterns();
       printf("%d\n",cost);
     }

     fclose(f);
     fclose(subs);
     fclose(pubs);
   }
   return 0;
}
```

EK 2

Bu ek bölümü ana koda eklenen "dataStructers.h" dosyasıdır.

```
#define MaxTask   110
#define MaxTag    30
#define MaxMol    20
#define MaxN        110
#define MaxM    110
#define ABS(a)    ((a)>0)?(a):(-(a))
#define Max(a,b)  (a<b)?(b):(a)
#define Min(a,b)  (a<b)?(a):(b)
#define us4(a)    (1<<(2*(a)))
#define getSubscribePack()  (struct SubscribePack
*)calloc(1,sizeof(struct SubscribePack))
#define getUpdatePack()   (struct UpdatePack
*)calloc(1,sizeof(struct UpdatePack))
#define subsPackVol  10
struct Molecule{  int no, X, Y; };
struct UpdatePack
{
  int sourceX, sourceY, molNo, newX, newY, volume, bitis;
  int targetX1, targetY1, targetX2, targetY2;
  int *used;
  struct UpdatePack *n;
};
struct SubscribePack
{
  int sourceNo, sourceX, sourceY, volume;
  int targetNo, targetX1, targetY1, targetX2, targetY2;
  int count, bitis;
  int mashx, mashy;
  int *used;
  struct SubscribePack *n;

};
struct Processor
{
  int no, X, Y, numMol, numRout;
  int routingTable[5000][7];// [0]->seviye, [1]->update
areaX1, [2]->update areaY1, [3]->update areaX2, [4]->update
areaY2, [5]->sonrakiX, [6]->sonrakiY
  struct Molecule molecules[MaxMol];
  struct UpdatePack *updates, *updtemp;
  struct SubscribePack *subscribes, *substemp;
  struct SubscribePack *mashSubQ[4];
  struct SubscribePack *mashSSon[4];
  struct UpdatePack *mashPubQ[4];
  struct UpdatePack *mashPSon[4];
};
```

ÖZGEÇMİŞ

Kimlik Bilgileri

Adı Soyadı : Nevzat SEVİM

Doğum Yeri : Adıyaman

Medeni Hali : Bekar

E-posta : nevzatsevim@gmail.com, nevzat.sevim@tubitak.gov.tr

Adresi : TÜBİTAK Başkanlık Tunus Caddesi No:80 06100 Kavaklıdere/ANKARA

Eğitim

Lisans : Hacettepe Üniversitesi – Bilgisayar Mühendisliği

Yüksek Lissans : Hacettepe Üniversitesi – Bilgisayar Mühendisliği

Yabancı Dil

İngilizce

İş Deneyimi

Ekim 2013 - ... TUBİTAK-Bilişim Müdürlüğü

Deneyim Alanları

Ağ Yönetimi, Linux ve Windows Sunucu Yönetimi, Sanallaştırma Yönetimi, Sistem Güvenlik Sorumlusu,

Tezden Üretilmiş Projeler ve Bütçesi

Tezden Üretilmiş Yayınlar

Kayhan İmre, Nevzat Sevim, "The High Level Architecture (HLA) on Photonic Torus: Hardware and Software Co-design" *8 th EUROSIM Congress*, Cardiff, Wales, United Kingdom, 10-13 September **2013**

Tezden Üretilmiş Tebliğ ve/veya Poster Sunumu ile Katıldığı Toplantılar

8'inci EUROSIM konferansında sunulmuştur.