KOŞUT ALGORİTMALARIN KOŞUT HESAPLAMA PLATFORMLARINA ATANMASI İÇİN MODEL GÜDÜMLÜ YAZILIM GELİŞTİRME

MODEL-DRİVEN SOFTWARE DEVELOPMENT FOR MAPPİNG OF PARALLEL ALGORİTHMS TO PARALLEL COMPUTİNG PLATFORMS

ETHEM ARKIN

YARD. DOÇ. DR. KAYHAN İMRE

Tez Danışmanı

PROF. DR. BEDİR TEKİNERDOĞAN

İkinci Tez Danışmanı

Hacettepe Üniversitesi

Lisansüstü Eğitim – Öğretim ve Sınav Yönetmeliğinin

Bilgisayar Mühendisliği Anabilim Dalı için Öngördüğü

DOKTORA TEZİ olarak hazırlanmıştır.

2015

ETHEM ARKIN'ın hazırladığı "**Koşut Algoritmaların Koşut Hesaplama Platformlarına Atanması İçin Model Güdümlü Yazılım Geliştirme**" adlı bu çalışma aşağıdaki jüri tarafından **BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**'nda **DOKTORA TEZİ** olarak kabul edilmiştir.

Prof. Dr. Ali Hikmet DOĞRU
Başkan ……

Yard. Doç. Dr. Kayhan İMRE
Danışman ……

Doç. Dr. Harun ARTUNER
Üye ……

Yard. Doç. Dr. Hasan SÖZER
Üye ……

Yard. Doç. Dr. Ayça TARHAN
Üye ……

Bu tez Hacettepe Üniversitesi Fen Bilimleri Enstitüsü tarfından **DOKTORA TEZİ** olarak onaylanmıştır.

Prof.Dr. Fatma SEVİN DÜZ
Fen Bilimleri Enstitüsü Müdürü

Eşim Ferda, Oğullarım Demirkan ve Doruk'a ...

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.

04/09/2015

ETHEM ARKIN

ÖZET

KOŞUT ALGORİTMALARIN KOŞUT HESAPLAMA PLATFORMLARINA ATANMASI İÇİN MODEL GÜDÜMLÜ YAZILIM GELİŞTİRME

Ethem ARKIN

Doktora, Bilgisayar Mühendisliği Bölümü

Tez Danışmanı: Yard. Doç. Dr. Kayhan İMRE

İkinci Tez Danışmanı: Prof. Dr. Bedir TEKİNERDOĞAN

Ağustos 2015, 111 sayfa

Mevcut eğilim bilgisayar sistemlerindeki kullanılan işlemci sayısının önemli ölçüde artışta olduğunu göstermektedir. 2020 yılı itibari ile super bilgisayarların yüksek ölçekli seviyede hesaplama yapabilmesi için yüzbinlerce işlem biriminden oluşması planlanmaktadır. Tek işlemciden koşut bilgisayar mimarilerine uzanan bu eğilim ile birlikte yüksek başarımlı hesaplama için gereken ihtiyaç koşut hesaplamanın benimsenmesini sağlamaktadır. Koşut hesaplama gücünden yararlanmak için bu koşut hesaplama platformlarına atanabilen ve çalıştırılabilen koşut algoritmaların tanımlanması gerekmektedir. Sınırlı sayıda işlem biriminden oluşan küçük hesaplama platformları için koşut algoritmaların atanması elle yapılabilmektedir. Ancak, yüksek ölçekli sistemler gibi büyük koşut hesaplama platformlarında olabilecek atama seçenek sayısı önemli ölçüde artmaktadır ve atama işlemi takip edilemez olmaktadır. Bu nedenle, uygun atamaların elde edilmesi ve hedef kodun oluşturulması için otomatik bir yaklaşımın tanımlanması gerekmektedir. Bu tezde, koşut algoritmaların koşut hesaplama platformlarına atanması için model güdümlü yazılım geliştirme yaklaşımı sunulmaktadır.

Yaklaşım, algoritma parçalamasının ve koşut hesaplama platformlarının modellenmesi, algoritmanın koşut hesaplama platformuna atanmasının modellenmesi için gereken yeniden kullanılabilir varlıkların tanımlanması, uygun atamaların oluşturulması ve son kodun üretilmesi ve konuşlandırılması için gereken faaliyetleri içermektedir. Modellemelerin yapılması için koşut hesaplama üst modeli tanımlanmıştır ve bu üst modelden mimari bakış açıları elde edilmiştir. Yaklaşım iyi bilinen koşut algoritmalar kullanılarak değerlendirilmiştir.

Anahtar Kelimeler: Koşut işlem, aşırı ölçekli sistemler, model güdümlü yazılım geliştirme, kod üretimi, mimari bakış açıları.

ABSTRACT

MODEL-DRIVEN SOFTWARE DEVELOPMENT FOR MAPPING OF PARALLEL ALGORITHMS TO PARALLEL COMPUTING PLATFORMS

Ethem ARKIN

Doctor of Philosophy, Deparment of Computer Engineering

Supervisor: Asst. Prof. Dr. Kayhan İMRE

Co-supervisor: Prof. Dr. Bedir TEKİNERDOĞAN

August 2015, 111 pages

The current trend shows that the number of processors used for computer systems are dramatically increasing. By the year 2020, it is planned that supercomputers will have hundreds of thousands of processing units to compute at exascale level. The need for high performance computing together with this trend from single processor to parallel computer architectures has leveraged the adoption of parallel computing. To benefit from parallel computing power usually parallel algorithms are defined that can be mapped and executed on these parallel computing platforms. For small computing platforms with a limited number of processing units the mapping process can be carried out manually. However, for large scale parallel computing platforms such as exascale systems, the number of possible mapping alternatives increases dramatically and the mapping process becomes intractable. Therefore, an automated approach to derive feasible mappings and generate target code must be defined. In this thesis a model driven software development approach for mapping parallel algorithms to parallel computing platforms is provided.

The approach includes several activities for modeling the algorithm decomposition and parallel computing platform, defining the reusable assets for modeling the

mapping of the algorithm to parallel computing platform, generating feasible mappings, and generating and deploying the final code. For modeling to be possible, the metamodel for the parallel computing is defined and architecture viewpoints are adopted from the metamodel. The approach is evaluated using well-known parallel algorithms.

Keywords: Parallel processing, exascale systems, model-driven software development, code generation, architecture viewpoints.

TEŞEKKÜR

Yazar, bu çalışmanın gerçekleşmesinde katkılarından dolayı, aşağıda adı geçen kişi ve kuruluşlara içtenlikle teşekkür eder.

Yard. Doç. Dr. Kayhan İmre (Tez Danışmanı) tez çalışmasının gerçekleştirilmesi için gerekli ortamın hazırlanmasında, çalışmanın sonuca ulaştırılmasında ve karşılaşılan güçlüklerin aşılmasında yön gösterici olmuştur.

Prof. Dr. Bedir Tekinerdoğan (İkinci Tez Danışmanı) tez çalışmasının gerçekleştirilmesi için gerekli ortamın hazırlanmasında, çalışmanın sonuca ulaştırılmasında ve karşılaşılan güçlüklerin aşılmasında yön gösterici olmuştur.

Doç. Dr. Harun Artuner ve Prof. Dr. Ali Hikmet Doğru tez çalışması sırasında yapılan çalışmaların takibini gerçekleştirmiş ve fikirleri ile yön gösterici olmuşlardır.

Eşim Hikmet Ferda ERGÜNEŞ ARKIN çalışmalarım sırasında sonsuz destek sağlamış ve oğullarım Demirkan ARKIN ve Doruk ARKIN ile birlikte motivasyon kaynağım olmuşlardır.

İÇİNDEKİLER

Sayfa

1. GİRİŞ 1

2. KOŞUT İŞLEM 5

2.1. KOŞUT HESAPLAMA GÜCÜ 5
2.2. KOŞUT ALGORİTMALAR VE KOŞUT MİMARİLER 6
2.3. KOŞUT BİLGİSAYARLAR 7
2.4. KOŞUT BİLGİSAYARLAR İÇİN HABERLEŞME AĞLARI 8
2.4.1. Ağ Topolojileri 9
2.5. KOŞUT PROGRAMLAMA 14
2.6. KOŞUT ALGORİTMA TASARIMI 15
2.6.1. Parçalama Teknikleri 15
2.6.2. Dağıtım Teknikleri 17
2.7. AŞIRI ÖLÇEKLİ KOŞUT SİSTEMLER 18

3. MODEL GÜDÜMLÜ YAZILIM GELİŞTİRME 21

3.1. GİRİŞ 21
3.2. MODELLEME 22
3.3. MODEL GÜDÜMLÜ GELİŞTİRME 23
3.3.1. Modelden Modele Dönüşüm 24
3.3.2. Modelden Metne Dönüşüm 26

4. KOŞUT İŞLEMİN MODELLENMESİ 27

4.1. KOŞUT İŞLEM MODELİ 28
4.1.1. Döşemeler 30
4.1.2. İletişim Örüntüleri 35
4.1.3. Çalıştırma Bölümleri 40
4.2. MODEL GÜDÜMLÜ YAKLAŞIM 44
4.2.1. Koşut Üst Model 46
4.2.2. Model Güdümlü Yazılım Geliştirme Yaklaşımı 48

5. ARAÇ DESTEĞİ 81

6. ÖRNEK ÇALIŞMALAR 84

6.1. VEKTÖREL TOPLAMA ALGORİTMASI 84
6.2. MATRİS ÇARPMA ALGORİTMASI 86
6.3. MATRİS DEVRİĞİ ALGORİTMASI 89
6.4. TÜMÜNÜ EŞLEME ALGORİTMASI 92

7. DEĞERLENDİRME 96

7.1. MANTIKSAL KONFİGÜRASYONLARIN OLUŞTURULMA PERFORMANSI 96
7.2. ÇOK ÇEKİRDEKLİ BİR PLATFORMDA ÇALIŞTIRMA 97
7.3. BENZETİM PERFORMANSI 98

8. SONUÇ 102

ŞEKİLLER

Sayfa

Şekil 1 Veri düzeyi koşutluk 6
Şekil 2 Çoklu komut çoklu veri koşut bilgisayar. (a) Dağıtılmış bellekli ve (b) paylaşımlı bellekli 8
Şekil 3 Haberleşme ağlarının sınıflandırılması: (a) Durağan ağlar ve (b) devingen ağlar [15]. 9
Şekil 4 Veri yolu tabanlı ağlar: (a) Önbelleksiz ağlar ve (b) yerel bellek/önbellekli ağlar. 10
Şekil 5 Birbirini engellemeyen çapraz ağlar. 10
Şekil 6 Çok safhalı ağlar [15]. 11
Şekil 7 Yıldız bağlantılı ağlar: (a) tamamen bağlı ağ ve (b) yıldız bağlı ağ [15]. .. 12
Şekil 8 Doğrusal diziler: (a) sarmal olmayan ve (b) sarmal bağlı [15]. 12
Şekil 9 İki ve üç boyutlu örgüler: (a) sarmal olmayan iki boyutlu örgü, (b) sarmal bağlı iki boyutlu örgü ve (c) sarmal olmayan üç boyutlu örgü [15]. 13
Şekil 10 Hiper küplerin oluşturulması [15]. 13
Şekil 11 Quicksort sıralama algoritmasının özyineli parçalama ile görev atamalarının belirlenmesi. 16
Şekil 12 Keşifsel olarak her ağaç aramasının ayrı işlemcilere parçalanması 17
Şekil 13 Modelleme kod ilişkileri. 22
Şekil 14 Model dönüşümü kavramsal görünümü. 24
Şekil 15 Model dönüşümü. 25
Şekil 16 Koşut dillerin sınıflandırılması. 28
Şekil 17 4x4 torus topolojisinin farklı gösterimleri: (a) Geleneksel gösterim, (b) gölgeli kare gösterim, (c) açık kare biçimi, (d) hakim düğümler ile soyut gösterim ve (e) hakim düğümler olmadan soyut gösterim. 30
Şekil 18 Farklı boyutlar için döşeme gösterimleri. 31
Şekil 19 16x16 torus için döşeme örnekleri: (a) Özyineli döşeme ve (b) tekrarlayıcı döşeme. 32
Şekil 20 7-işlemcili bir döşemeden özyineli olarak 49-işlemcili döşemenin oluşturulması. 32
Şekil 21 Strassen'in koşut sisteminin özyineli parçalanması. 33
Şekil 22 Çeşitli boyutta yapıların oluşturulması için özyineli ve tekrarlayıcı döşemeler: (a) 5x5 ve 4x4 döşemelerden 20x20 döşemenin oluşturulması, (b) 4x4 ve 2x2 döşemelerden 8x8 döşemenin oluşturulması, (c) (d) 4x4 ve 7x7 döşemelerden 28x28 döşemenin oluşturulması. 34
Şekil 23 Örüntüler ve uygun iletişim yollarından oluşturulan iletişim örüntüleri: (a) Hakim düğümler arasındaki iletişim örüntüsü ve (b) Hakim düğümlerden döşeme içindeki diğer düğümlere iletişim örüntüsü. 35
Şekil 24 4 adet T-şekilli döşeme ile oluşturulmuş 4x4 döşemesi için birinden-hepsine algoritması iletişim örüntüsü. 36
Şekil 25 Özyineli ölçeklendirilmiş 16x16 döşeme için birinden-hepsine iletişim örüntüsü. 37
Şekil 26 4x4 döşeme kullanarak tüm-değişim algoritması. 40
Şekil 27 4x4 torus üzerinde tüm-değişim örneğinin çalıştırılması için sıralı bölümler 41
Şekil 28 16x16 torus için tüm-değişim algoritması (ilk üç adım). 42
Şekil 29 16x16 torus için tüm-değişim algoritması (son üç adım). 43

Şekil 30 Koşut sistemler için model-güdümlü dönüşüm zinciri. 45
Şekil 31 Model dönüşümü kavramsal gösterimi. 45
Şekil 32 Koşut işlem üst modeli. 47
Şekil 33 Koşut algoritmaların koşut hesaplama platformlarına atanması için model güdümlü yazılım geliştirme yaklaşımı. 49
Şekil 34 Fiziksel Konfigürasyon Görünümü 52
Şekil 35 Örnek *Gather* ve *Scatter* işlevleri döşeme ve iletişim örüntüleri. 53
Şekil 36 Mantıksal konfigürasyonun oluşturulması algoritması. 56
Şekil 37 4x4 Fiziksel konfigürasyon için iki farklı iletişim örüntüsü kullanılarak oluşturulmuş mantıksal konfigürasyonlar. 57
Şekil 38 MPI üstmodeli. 60
Şekil 39 MPI veri üstmodeli. 60
Şekil 40 MPI işlem maddeleri üstmodeli. 61
Şekil 41 MPI noktadan noktaya işlem üstmodeli. 62
Şekil 42 MPI tek yönlü toplu işlem üstmodeli. 62
Şekil 43 MPI çift yönlü toplu işlem üstmodeli. 62
Şekil 44 MPI durum (*state*) işlemleri üstmodeli. 63
Şekil 45 Yongaya bağlı ağ bileşeni kavramsal gösterimi. 64
Şekil 46 NoC yongalar için temel ayarlar. 65
Şekil 47 Tüm-değişim algoritması için NoC durumlarının gösterimi. 65
Şekil 48 Düğümün özyineli parçalanması sonucu oluşan alt durumların gösterimi. Doğudan güneye bir durumun alt durumları gösterilmektedir. 66
Şekil 49 NoC üstmodeli. 68
Şekil 50 Koşut modelden MPI modeline dönüşüm kuralları. 69
Şekil 51 Alt parçaların oluşturulması için yardımcı bağlam. 69
Şekil 52 Son işlem birimi kontrolü yapan ve tüm işlem birimlerini bulan yardımcı bağlam. 70
Şekil 53 Tüm-değişim örneği için dönüştürülen MPI modeli. 71
Şekil 54 Koşut modelden NOC modeline dönüşüm kuralları. 72
Şekil 55 NOC sıralama tanımlamaları. 73
Şekil 56 Hakim düğümün pozisyonuna göre NOC ayarlarını oluşturan yardımcı bağlam. 74
Şekil 57 NOC ayarlarının dönüşümü. 75
Şekil 58 Tüm-değişim örneği için dönüştürülen NoC modeli. 76
Şekil 59 cTypeDefinition şablonu. 77
Şekil 60 typeDefinition şablonu. 77
Şekil 61 mpiDataTypeDefinition şablonu. 78
Şekil 62 mpiTypeDefinition şablonu. 78
Şekil 63 mpiDerivedTypeDefinition şablonu. 78
Şekil 64 mpiDerivedTypeOperation şablonu. 79
Şekil 65 mpiOperationDefinition şablonu. 79
Şekil 66 MPI modelden MPI kod dönüşümü şablonu. 80
Şekil 67 Parmapper Aracı Kavramsal Tasarımı. 82
Şekil 68 Parmapper Kütüphane Tanımlama Aracı Ekran Görüntüsü. 83
Şekil 69 Parmapper Algoritma Atama Aracı Ekran Görüntüsü. 83
Şekil 70 Vektörel toplama algoritması. 84
Şekil 71 Vektörel toplama algoritması mantıksal konfigürasyon görünümü 85
Şekil 72 Vektörel toplama algoritması koşut modeli 86
Şekil 73 Matris çarpma algoritması. 87
Şekil 74 Matris çarpma algoritması mantıksal konfigürasyon görünümü 88

Şekil 75 Matris çarpma algoritması koşut modeli 89
Şekil 76 Matris devriği algoritması. 89
Şekil 77 Matris devriği algoritması mantıksal konfigürasyon görünümü 91
Şekil 78 Matris devriği algoritması koşut modeli 91
Şekil 79 Tümünü eşleme algoritması. 92
Şekil 80 Tümünü eşleme algoritması mantıksal konfigürasyon görünümü (ilk üç adım). 94
Şekil 81 Tümünü eşleme algoritması mantıksal konfigürasyon görünümü (son üç adım). 95
Şekil 82 Üretilen koşut MPI kodları için benzetim ortamı. 99
Şekil 83 Tümünü eşleme algoritması için görsel gösterim. 99
Şekil 84 Vektörel toplama algoritması için iletişimin hızlanması karşılaştırması. 100
Şekil 85 Matris çarpma algoritması için iletişimin hızlanması karşılaştırması. 100
Şekil 86 Matris devriği algoritması için iletişimin hızlanması karşılaştırması. 101
Şekil 87 Tümünü eşleme algoritması için iletişimin hızlanması karşılaştırması. . 101

ÇİZELGELER

Sayfa

Çizelge 1 Hakim düğümler ile yönetilen düğümler arasındaki temel toplu işlemler 38
Çizelge 2 Algoritmalar 39
Çizelge 3 Algoritma Parçalama Bakış Açısı 50
Çizelge 4 Algoritma Parçalama Görünümü 50
Çizelge 5 Fiziksel Konfigürasyon Bakış Açısı 51
Çizelge 6 Algoritmadan-Mantıksal Konfigürasyona Atama Bakış Açısı 54
Çizelge 7 Algoritmadan-Mantıksal Konfigürasyona Atama Görünümü 55
Çizelge 8 Mantıksal Konfigürasyon Bakış Açısı 56
Çizelge 9 Alt düğümdeki hakim düğüme gore komşularının durumları çizelgesi. . 67
Çizelge 10 Vektörel toplama algoritması parçalama görünümü 84
Çizelge 11 Vektörel toplama algoritmasından mantıksal konfigürasyona atama görünümü 85
Çizelge 12 Matris çarpma algoritması parçalama görünümü 87
Çizelge 13 Matris çarpma algoritmasından mantıksal konfigürasyona atama görünümü 88
Çizelge 14 Matris devriği algoritması parçalama görünümü 90
Çizelge 15 Matris devriği algoritmasından mantıksal konfigürasyona atama görünümü 90
Çizelge 16 Tümünü eşleme algoritması parçalama görünümü 92
Çizelge 17 Matris devriği algoritmasından mantıksal konfigürasyona atama görünümü 93
Çizelge 18 Mantıksal konfigürasyonların oluşturulma performansı 97
Çizelge 19 Algoritmaların çok çekirdekli platformda çalıştırılmasında hesaplanan metrikler ve ölçülen değerler 98

SİMGELER VE KISALTMALAR

FLOPS : Saniyede yapılan kayan noktalı işlem (*floating point operations per second*) sayısı. Bu birim bilgisayar sistemlerinin performansının belirtilmesinde kullanılan birimdir.

MDA : Model Güdümlü Mimari (*Model Driven Architecture*)

MGYG : Model Güdümlü Yazılım Geliştirme

MIMD : Çoklu Komut Çoklu Veri (*Multiple Instruction Multiple Data*)

MISD : Çoklu Komut Tek Veri (*Multiple Instruction Single Data*)

MOF : Meta-Object Facility

MPI : Message Passing Interface

OMG : Object Management Group

PIM : Platformdan Bağımsız Model (*Platform Independent Model*)

PM : Platform Modeli

PSM : Platforma Bağlı Model (*Platform Specific Model*)

SIMD : Tek Komut Çoklu Veri (*Single Instruction Multiple Data*)

SISD : Tek Komut Tek Veri (*Single Instruction Single Data*)

UML : Unified Modeling Language

1. GİRİŞ

Günümüzde bilgisayar sistemleri tek işlemcili sistemlerden çok işlemci içeren sistemlere doğru gelişmektedir. İşlemcilerin fiziksel olarak sınırlarına dayandığı [1] gözönüne alındığında daha çok başarım gerektiren bilgisayar sistemleri için koşut bilgisayar mimarileri kullanılması gerektiği ortaya çıkmaktadır. Bunun sonucu olarak da ölçeği sürekli olarak artan çok sayıda işlem birimi barındıran hesaplama platformları oluşturulmaya başlanmıştır. 2020 yılı itibari ile de yüksek ölçekli sistemlerin kurulması planlanmaktadır [2].

Koşut bilgisayar sistemleri daha büyük, daha hızlı ve kullanımı daha yaygın olmaya başladıkça, seri programlama ile çözülmesi çok uzun süren problemlerin çözümü için daha müsait olmaya başlamıştır. Bilim, mühendislik ya da ticari uygulamalar gibi yüksek başarımlı hesaplama gerektiren çeşitli alanlarda koşut işlem kullanılabilmektedir. Koşut bir programın geliştirilmesi için, problemin birbirinden ayrı parçalara parçalanması, tüm parçaların koşut hesaplama platformundaki işlem birimlerine atanması, bu işlem birimlerinde yerel olarak işlenmesi ve sonuçlarının birleştirilerek çözümün oluşturulması gerekmektedir. Ayrık parçalara ayırma, ya da diğer bir adı ile parçalama (*decomposition*), iki aşamadan oluşmaktadır: öncelikle problemin işlem birimlerine atanması, sonrasında problemin işlem birimlerine dağıtılması [3]. Dağıtım safhası, haberleşme ek yükünün küçültülmesi ve koşutluğun arttırılması arasında dengelemek için kararları içerdiği için oldukça karmaşıktır. Ayrıca işlemciler arası haberleşme de koşut sistemin performansını etkiler [4]. Koşut sistem büyüdükçe ve dağıtım olasılıkları arttıkça, koşut programcının sistem performansını büyütmek için problemi parçalamanın seçimi zorlaşmaktadır.

Sistemin etkinliği aynı zamanda uyuşmazlıktan bağımsız ve kilitlenmeden bağımsız iletişimlere bağlıdır. Kilitlenmeyi engelleme ve kilitlenmeden sakınma gevşek bağlı koşut mimarilerde sık karşılaşılan bir problemdir [5]. Ayrıca paylaşımlı ağ topolojilerinde uyuşmazlıklar adresleme stratejileri için sorun teşkil etmektedirler [6]. Programcı programın davranışını, bellek ve hesaplama kısıtlarını ve parçalama probleminde kullanılacak iletişim yollarını değerlendirmelidir. Programlama modeli bağımlılıkları, çelişkileri, kilitlenmeleri, uyuşmazlıkları ve diğer sorunları denetleyerek programcının programın doğruluğu hakkında emin olmasını sağlamalıdır [7].

Koşut programlama dilleri incelendiğinde, araçlar ve ortamlar koşut program tasarımı ve gerçekleştirme için gerekli şeylerin başında gelmektedir [8]. Koşut yazılım geliştirme yöntemleri dört önemli meseleyi adresler: problemin nasıl parçalanacağı, faaliyetlerin nasıl birlikte çalışacağı, yapıyı ve hesaplama maliyetlerini değerlendirmek amacıyla nasıl genel olarak görülebileceğini ve hesaplamaların işlemcilere nasıl atanacağı. Problem alanının parçalanması ve faaliyetlerin birlikte çalışması programlama diline göre farklı biçimlerde gerçekleştirilebilmektedir [8]. Bu durum programcının koşut programlama diline göre algoritmalarını tasarlamasını zorunlu kılar. Araştırmalar eğilimlerin düşük seviye programlama dillerinden, mimari ile ilgili sorunları kullanıcıdan saklayan ve koşutluk, programın parçalanması, iletişim, eşgüdümleme ve eşleştirme gibi problemlere yönelen daha soyut dillere doğru kaydığını göstermektedir [8]. Koşut hesaplama için uygulanabilir birleşik modeller (*unified models*) son yıllarda daha çok çalışılmaya başlanmıştır ve koşut programlamanın geleceğinde önemli bir rol oynayacağı öngörülmektedir. Model güdümlü yazılım geliştirme yöntemi daha soyut diller ve birleşik hesaplama modelleri oluşturmak için iyi bir uygulamadır. Bu yöntem ile kullanılan modelleme, görselleştirme sağladığı için de tasarım safhasında önemli bir kazanç olmaktadır. Ayrıca görselleştirme koşut hesaplamanın karmaşıklığının yönetilmesinde yardımcı olmaktadır [9]. İşlemlerin koordinasyonu ve parçalanmasının modellenmesi görselleştirme için iyi bir yöntem olmaktadır. Tasarım safhasından sonra ise model güdümlü geliştirme yöntemleri ile koşut programların üretilmesi sağlanabilmektedir. Programların otomatik üretilmesi tasarım sürecinin hedeflenen programlam dili ile gerçekleştirme sürecinden ayırmakta ve bu yöntemin tüm koşut programlama dilleri için birleşik bir yöntem olmasını sağlamaktadır.

Modelleme gerçek sistemin daha soyut bir görünüme kavuşturan bir yöntemdir. Bu yöntem tasarımın yönetimini kolaylaştırır, uyuşmazlıkların, çelişkilerin ve diğer problemli meselelerin kontrol edilebilmesini sağlar ve tasarımcının tüm resmi genel olarak görmesine yardımcı olur. Tasarım aşaması sonrasında, bu modeller model-güdümlü yazılım geliştirme yöntemleri ile algoritmaların otomatik olarak oluşturulmasında kullanılabilirler. Ayrıca kaynak kodların model-güdümlü olarak otomatik oluşturulması oluşabilecek kodlama hatalarının önüne geçer ve kodun daha az hatalı olmasını sağlar.

Model-güdümlü yazılım geliştirme yöntemi daha soyut bir dil ve birleştirilmiş hesaplama modeli oluşturmak için iyi bir uygulamadır. Ayrıca modelleme ile görselleştirme de tasarım aşamasında anahtar bir kazanım olmaktadır. Koşut işlemde gerçekleştirilen eşgüdüm ve parçalama işlemlerinin modellenmesi, işlemin görselleştirilmesini sağlamaktadır. Tasarım safhasından sonra oluşturulan modellerin koşut algoritmaların hedeflenen programlama diline uygun olarak otomatik üretilmesinde kullanılması, tüm koşut programlama dilleri için birleştirilmiş bir yöntem oluşturulmasını sağlamaktadır.

Bu tez kapsamında geliştirilen yaklaşımda ise, işlemlere ait atamalar modellenmekte ve küçük ölçekte yönlendirme stratejisi belirlenmektedir. Bu atamalar belirlendikten sonra işlem örüntüleri tasarlanmaktadır. Küçük ölçekte hazırlanan modeller tez içinde açıklanan koşut gösterime uygun olarak daha büyük boyutlara ölçeklendirilmektedir. Koşut algoritmaların modellenmesi için aşağıdaki adımlar gerçekleştirilmektedir:

- Algoritmanın adımlarının (kesimlerinin) belirlenmesi,
- Fiziksel hesaplama platformu öğelerinin belirlenmesi,
- Algoritmaya uygun atama planlamasının yapılması (iletişim örüntüleri ve nasıl ölçekleneceği),
- Ölçekleme ile atama seçeneklerinin üretilmesi,
- Uygun atamanın seçilmesi,
- Seçilen atamaya uygun olarak kod üretiminin gerçekleştirilmesi.

Bu adımlar sırasında oluşturulan modeller fizksel hesaplama platformunun büyümesi durumunda iletişim örüntülerine uygun olarak tüm atama seçeneklerini görselleştirebilmektedir. Bu da Talia [8] tarafından belirtilen koşut programlar için ana sorunlardan biri olan, geliştiricinin büyük topolojiler üzerinde algoritmanın davranışını genel olarak görmesini sağlamaktadır.

Bu tezde koşut algoritmaların koşut hesaplama platformlarına atanabilmesi için model güdümlü yazılım geliştirme yöntemi anlatılmaktadır. Bu yaklaşım ile koşut

algoritmaların koşut hesaplama platformlarına atanması için kod üretimi model güdümlü yaklaşım ile yapılmaktadır. Koşut programcıların problemi döşeme yöntemi kullanarak parçalaması, iletişim örüntüleri ile iletişimlerin eşgüdümlenmesi ve hesaplamaların yapılacağı işlemlerin tanımlanması sağlanmıştır. Bu kapsamda koşut programlama için üst model ve bu üst modelden elde edilen mimari bakış açıları tanımlanmıştır. Bu mimari bakış açıları kullanılarak oluşturulan mimari görünüm modelleri ile tasarım ve otomatik kod üretim yöntemleri açıklanmıştır.

Tezin birinci bölümünde tez konusu ve yapılan çalışmalar ile ilgili bilgiler sunulmuştur. İkinci bölümde koşut işlem, koşut işlemin gelişimi, koşut mimariler, ağ topolojileri, algoritma tasarımı ile ilgili teknikler ve aşırı ölçekli koşut sistemler anlatılmıştır. Üçüncü bölümde model güdümlü yazılım geliştirme, modelleme, modelden modele ve modelden metne dönüşüm konularına yer verilmiştir. Dördüncü bölümde tez kapsamında oluşturulan koşut algoritmaların koşut hesaplama platformlarına atanması için model güdümlü yazılım geliştirme yaklaşımı anlatılmıştır. Beşinci bölümde tez çalışmasını destekleyen araç, altıncı bölümde ise tez kapsamında anlatılan yaklaşım ile hazırlanmış örnek çalışmalar gösterilmiştir. Örnek çalışmalar ile yapılan değerlendirmeler yedinci bölümde sunulmuştur. Sekizinci bölümde ise tez çalışmasının sonuçları anlatılmıştır.

2. KOŞUT İŞLEM

2.1. KOŞUT HESAPLAMA GÜCÜ

1965 yılında Gordon Moore [10] bir tümleşik devre üzerindeki bileşen sayısının her iki yılda bir ikiye katlanacağını belirtmiştir. Gordon Moore'un açıklaması cihaz karmaşıklığı ve zamana göre yaptığı deneysel gözlemlere dayanıyordu [11]. 1975 yılında ise bir tümleşik devre üzerinde aynı oranda bileşenin kullanılabilmesi bu kuralı destekler nitelikte olmuş ve bu kuramının 18 ay olacak biçimde güncellenmesini sağlamıştır. Bu kuram Moore Yasası olarak bilinmektedir.

Moore Yasası uzun yıllar boyunca bilgisayar yongalarının hızlarının sürekli olarak artması ve maliyetlerinin de düşmesi ile desteklendi. Ancak günümüzde silikon teknolojileri ile geliştirilen bu tümleşik devrelerin kullanılan transistör sayısının artırılması ile performanslarının artırılamayacağı görülmektedir. Bu sebeple bilgisayar sistemlerinin tek bir işlemci kullanılarak değil, birden fazla işlemci kullanılması yoluna gidilmesi kaçınılmaz olacaktır.

Tüm hesaplama hızı aslında sadece işlemcinin hızına değil aynı zamanda verilerin beslenmesini sağlayan bellek sistemlerinin de hızına bağlıdır. İşlemci hızlarındaki hızlı artışa rağmen belleklerdeki hız artışı daha düşük seviyelerde kalmıştır. Özelikle disk erişim hızlarının oldukça düşük seviyelerde kalması tüm sistem performanslarının düşmesine sebep olmaktadır. Bu sebeple bir bilgisayar sisteminin hızı, içerdiği transistör sayısı ya da işlemcinin erişebildiği saat vurumu ile değil saniyede yaptığı işlem üzerinden değerlendirilmesi gerekmektedir. Bu sebeple saniyede yapılan kayan noktalı işlem (*floating point operations per second - FLOPS*) sistemin performansının ölçülmesi için kullanılmaktadır.

Tek bir işlemci yerine birden fazla işlemcinin koşut olarak kullanılması ile koşut sistemler oluşturulmaktadır. Koşut sistemlerde FLOPS değerleri yukarıda bahsedildiği gibi işlemcilerin hızı, bu işlemcilerin eriştiği bellek sistemlerinin hızı ve tek işlemcili sistemlerden farklı olarak aralarındaki haberleşme hızı ile hesaplanmaktadır. 2015 yılına gelinceye kadar çeşitli sistemler kurulmuş ve bu sistemler peta FLOPS düzeyine çıkmaktadır. 2015 yılı itibari ile geliştirilen "*MilkyWay-2*" koşut bilgisayar sistemi 33 peta FLOPS düzeyine erişmiştir ve bu sistem üzerinde 3,120,000 adet işlemci kullanılmıştır [12].

Koşut bilgisayar sistemleri birden çok işlemin koşut olarak işletilebilmesi amacıyla birden çok işlemciden oluşan sistemlerdir. Burada tasarıma bağlı olarak bir işlemci basit bir bileşen de olabilir, ya da çoklu kullanımlı işletim sistemi içeren işlemciler de olabilir.

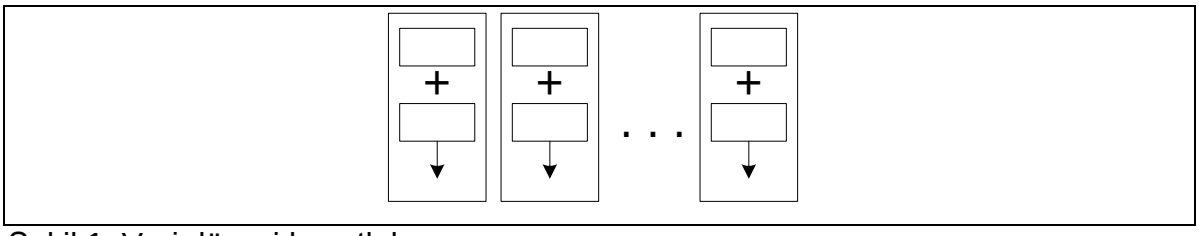
İşlemcilerin bir ağ bağlantısı kullanarak aralarında haberleşmesi gerekmektedir. Bir darboğazın oluşmaması için bu ağ bağlantısının farklı işlemci çiftleri arasında eş zamanlı haberleşmeye izin veriyor olması şarttır. Verinin nasıl değiştirileceği önceden tanımlanmalıdır. Bir veri yolu, ağ bağlantısı için en basit örnektir. Günümüzde ise veri yollarının yerini yonga üstü iletişim ağları (*network-on-chips, NOC*) almaktadır. Bu mimaride veri, yongalar arasında paketler olarak ve yönelticiler sayesinde iletilmektedir.

Veri ve programların bellek sistemlerinde saklanması gerekmektedir. Bellek birimleri işlemcilerin erişebileceği şekilde paylaştırılabilir ya da her işlemciye bir bellek birimi ayrılabilir. Veri işlemcilere paylaştırılacağı zaman her işlemci belli zamanlarda bu bellek bileşeni ile okuma ve yazma işlemi gerçekleştirir. Veri bütünlüğü için bu okuma ve yazma işleminin işlemciler arasındaki sırası önem kazanır.

2.2. KOŞUT ALGORİTMALAR VE KOŞUT MİMARİLER

Koşut algoritmalar ve koşut mimariler birbirlerine sıkı sıkıya bağlıdır. Bir koşut algoritmayı destekleyeceği donanım altyapısını düşünmeden oluşturamayız. Aynı şekilde bir koşut mimariyi üzerinde çalışacak algoritmalardan bağımsız tasarlamak da anlamsız olur. Koşutluk çeşitli seviyelerde donanım ve yazılım teknikleri kullanılarak yapılır [13]:

1. Veri Düzeyi Koşutluk: Eşzamanlı olarak çoklu veri üzerinde işlem yapılmasıdır. Örneğin ikili sayıların koşut olarak ikil seviyesi koşut toplama, çarpma ve bölme işlemleri ya da vektör dizilerin koşut toplanması verilebilir.



Şekil 1 Veri düzeyi koşutluk

2. Komut Düzeyi Koşutluk: Bir işlemcide eş zamanlı olarak birden fazla komutun işletilmesidir. Komut ardışık düzeni (*instruction pipeline*) örnek verilebilir.

3. İş Parçacığı Düzeyi Koşutluk: Bir iş parçacığı bir program içinde diğer iş parçacıkları ile işlemcinin kaynaklarını paylaşan birimdir. Bu düzeyde koşutluk için bir işlemci üzerinde farklı iş parçacıkları eş zamanlı olarak yürütülür.

4. İşlem Düzeyi Koşutluk: Bir işlem bir bilgisayarda çalışan program olarak tanımlanabilir. Bir işlem bilgisayardaki bellek alanı, yazmaç gibi kaynakları kullanır. Bu düzeydeki koşutluk birden fazla bilgisayarda çalışan işlemlerin eş zamanlı yürütülmesidir.

İlk üç koşutluk düzeyi tek işlemcili bilgisayarlarda kullanılan tekniklerdir ve günümüzde ihtiyaç sebebi ile birden fazla işlemcinin kullanıldığı koşut bilgisayarlarda işlem düzeyi koşutluk önem arz etmektedir.

2.3. KOŞUT BİLGİSAYARLAR

Bir koşut bilgisayarın tasarımı Flynn [14] tarafından veri ve işlemlerine göre şu şekilde sınıflandırılmıştır:

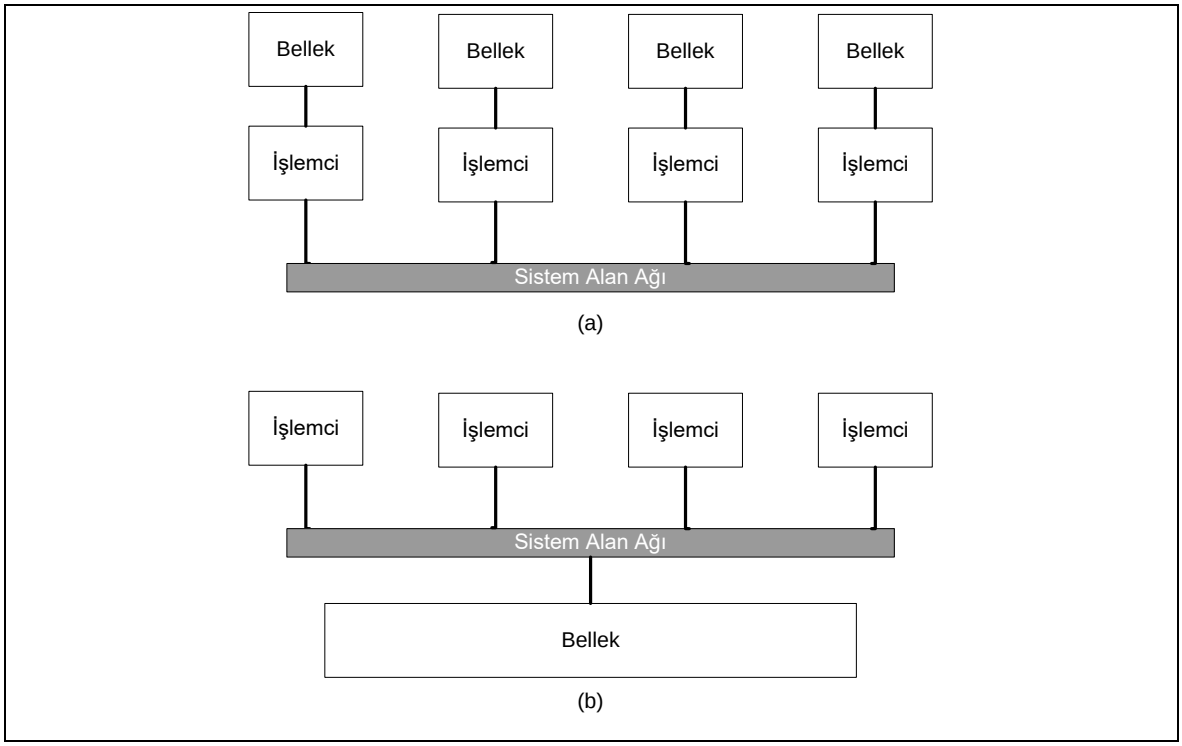
1. Tek Komut Tek Veri (*Single Instruction Single Data - SISD*): Bu tek bir işlemciden oluşan sistemdir.

2. Tek Komut Çoklu Veri (*Single Instruction Multiple Data - SIMD*): Tüm işlemcilerin farklı veriler üzerinde aynı komutları çalıştırdığı bilgisayarlardır. Her işlemci yerel belleğinde kendi verisini tutar. Buna örnek olarak grafik işlemcileri, video sıkıştırma ya da medical görüntü işleme verilebilir.

3. Çoklu Komut Tek Veri (*Multiple Instruction Single Data - MISD*): Yapay Sinir ağları ve veri akışı makineleri gibi bilgisayar sistemleridir.

4. Çoklu Komut Çoklu Veri (*Multiple Instruction Multiple Data - MIMD*): Her işlemcinin kendi komutlarını kendi verileri üzerinde çalıştırdığı bilgisayar sistemleridir. Günümüzde kullanılan çok çekirdekli işlemciler ve çoklu işlemli işlemciler örnek verilebilir.

Flynn'in sınıflandırması tüm bilgisayar sistemlerini koşutluk düşünüldüğünde kabaca bir sınıflandırmadır. Ancak koşut bilgisayarlar göz önüne alındığında günümüz bilgisayarlarının hep Çoklu komut Çoklu Veri sınıfına düştüğünü görüyoruz. Bu noktada gerek donanım olarak gerekse geliştirilecek olan koşut yazılım olarak Çoklu komut Çoklu Veri bilgisayarları dağıtılmış ve paylaşımlı bellekli olarak ikiye ayrılabilir.

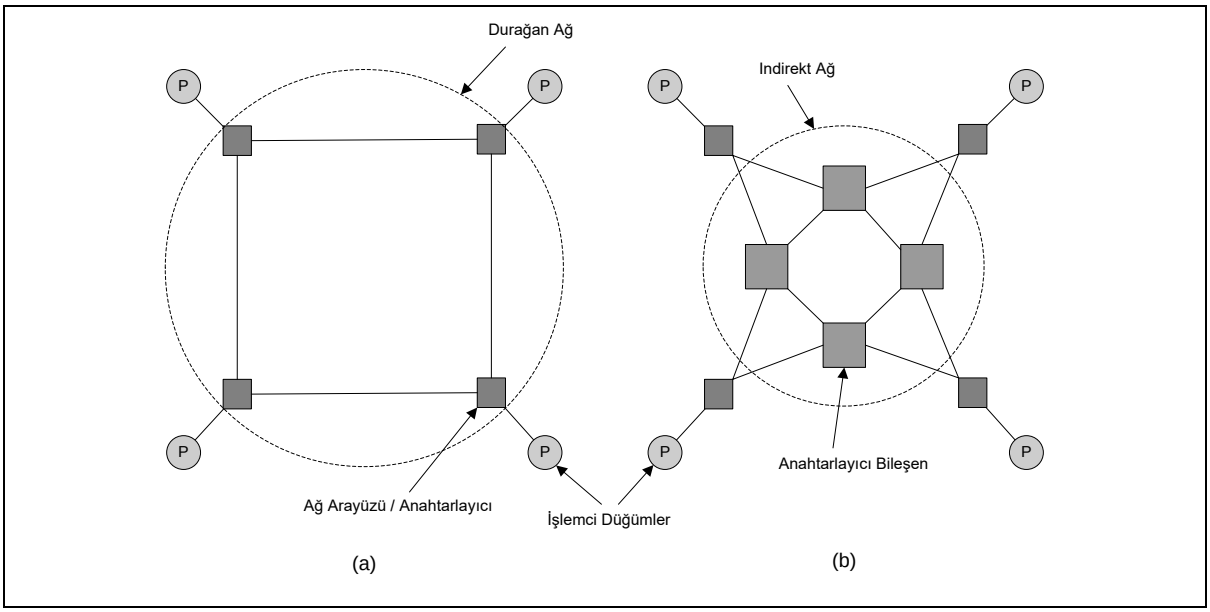


Şekil 2 Çoklu komut çoklu veri koşut bilgisayar. (a) Dağıtılmış bellekli ve (b) paylaşımlı bellekli

2.4. KOŞUT BİLGİSAYARLAR İÇİN HABERLEŞME AĞLARI

Haberleşme ağları, işlem birimleri arasındaki ya da işlemci ile bellek bileşenleri arasındaki veri alışverişini sağlayan altyapıları oluşturur. Tipik ağlar bağlantılar ve anahtarlayıcılar kullanılarak oluşturulurlar. Bir bağlantı veriyi taşıyan kablo ya da fiberden oluşan fiziksel ortamı ifade eder.

Haberleşme ağları durağan ya da devingen olarak iki sınıfa ayrılabilir [15]. Durağan ağlarda işlemciler noktadan noktaya bağlantılar ile ağı oluştururlar. Bu ağlarda verinin taşınacağı alternatif yollar az sayıdadır, ancak iletişim hızlı olarak gerçekleştirilir. Devingen ağlarda ise bağlantılar ağ anahtarlayıcıları ile oluşturulur. Bu yapıda ise birçok alternatif iletişim yolu oluşturulabilirken, verinin gideceği yolun gerçek zamanlı olarak bulunması gerektiği için hızı daha yavaş olabilmektedir.



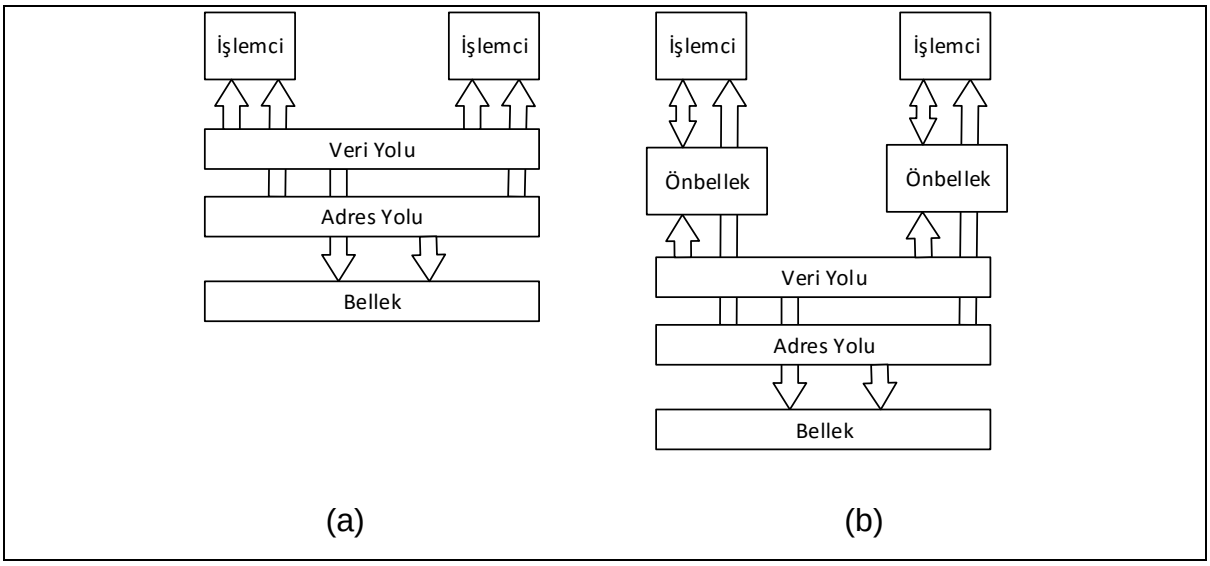
Şekil 3 Haberleşme ağlarının sınıflandırılması: (a) Durağan ağlar ve (b) devingen ağlar [15].

2.4.1. Ağ Topolojileri

Koşut bilgisayar sistemlerinde kullanılan haberleşme ağları performansa bağlı olarak maliyet ve ölçeklendirilebilirliğe göre oluşturulmaktadır. Bu ağ topolojileri bu bölümde incelenmiştir.

2.4.1.1. Veri Yolu Tabanlı Ağlar

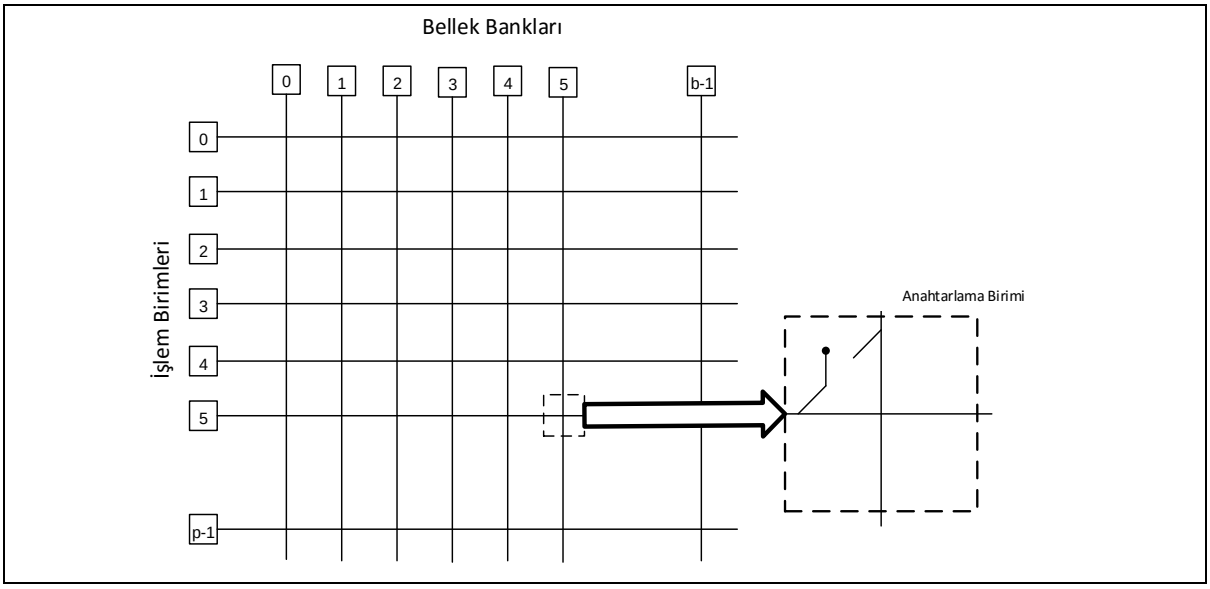
Bu ağ topolojisi en basit ağ topolojisidir. Bu topolojide ortak bir ağ ortamı tüm işlemci bileşenlerine erişilebilirdir. Bu sayede bu ortak ağ ortamı üzerinden işlemci bileşenleri haberleşirler. Ayrıca bu ağ ortamı yayımlama haberleşmesi için de çok uygundur. Ancak noktadan noktaya haberleşmelerde düğüm sayısı arttıkça haberleşmedeki maliyet ve bu ortamın paylaşım sıklığı artmaktadır. Ağ bant genişliği sorununu kısmen çözmek için yerel ön bellekler kullanılabilir.



Şekil 4 Veri yolu tabanlı ağlar: (a) Önbelleksiz ağlar ve (b) yerel bellek/önbellekli ağlar.

2.4.1.2. Çapraz Ağlar

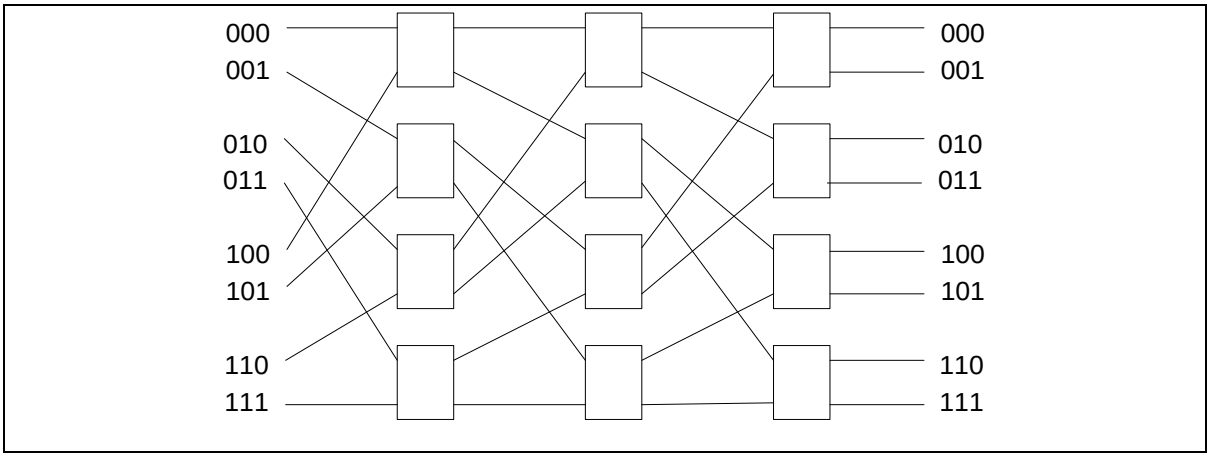
Bu topoloji özellikle işlemci birimlerini bellek bileşenlerine anahtarlamakta kullanılmaktadır. Çapraz ağ topolojisi anahtarlayıcı ızgaraları ya da anahtarlayıcı düğümleri kullanır. Birimler arası haberleşmeler diğer haberleşmeleri engellemez. Ancak bir birim aynı anda sadece tek bir birim ile iletişimde bulunabilir. Anahtarlama durumlarının da ayrıca yönetilmesi gerekir.



Şekil 5 Birbirini engellemeyen çapraz ağlar.

2.4.1.3. Çok Safhalı Ağlar

Çapraz ağlar performans açısından etkin bir topoloji olmasına rağmen ölçeklendirilemeyen ağ topolojilerindendir. Bunun yanında veri yolu tabanlı ağlar da ölçeklendirilebilir ancak performans açısından etkin olmayan bir topolojidir. Çok safhalı ağlar ise bu iki ağ topolojisinin arasında yer alır.



Şekil 6 Çok safhalı ağlar [15].

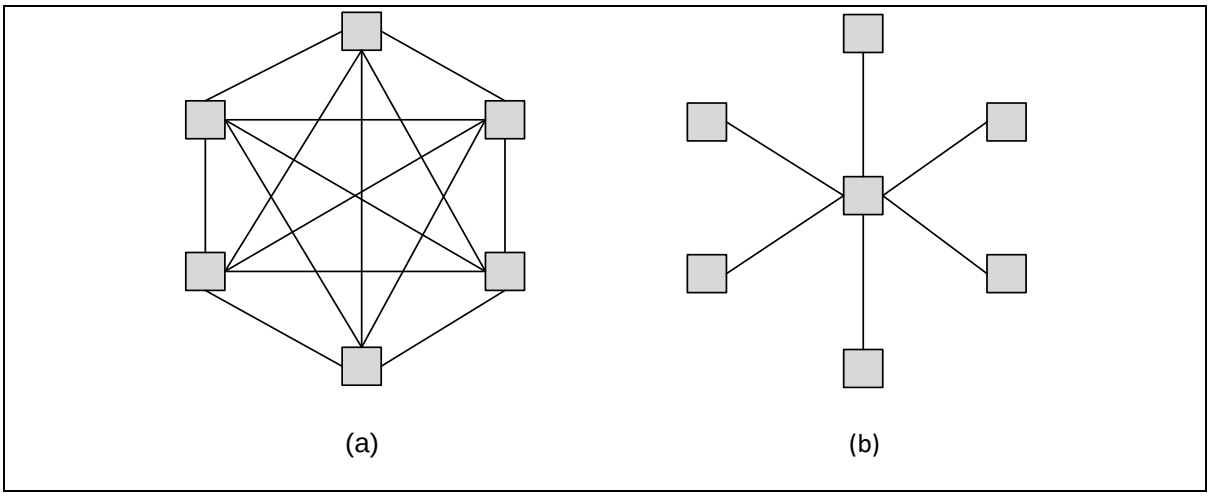
Çok safhalı ağların her aşamasında o anahtarlayıcıya ait girdilerin çıktılarına nasıl bağlanacağı bir örüntü ile belirlenir.

2.4.1.4. Doğrudan Bağlı Ağlar

Doğrudan bağlı ağlarda ağ üzerinde bulunan her düğümün diğer tüm düğümlere doğrudan bir haberleşme bağlantısı mevcuttur. Bu ağ topolojisi çapraz ağlar ile neredeyse aynı yapıdadır, sadece hangi bağlantının kurulacağının yönetilmesine gerek yoktur ve bir bağlantı diğer bir bağlantıyı etkilemez. Ancak performans açısından etkin olmasına rağmen ölçeklendirilemeyen ağ topolojilerindendir.

2.4.1.5. Yıldız Bağlantılı Ağlar

Yıldız bağlantılı ağlarda bir düğüm merkezi düğüm rolünü üstlenir. Diğer tüm düğümlerin bu merkezi düğüme haberleşme bağlantısı vardır. Bu ağ topolojisi veri yolu tabanlı ağlara benzer. Herhangi iki düğüm arasındaki haberleşme merkezi düğüm üzerinden gerçekleştirilir, bu durum veri yolu tabanlı ağlardaki ortak ortamın paylaşılması ile aynı durumdur.

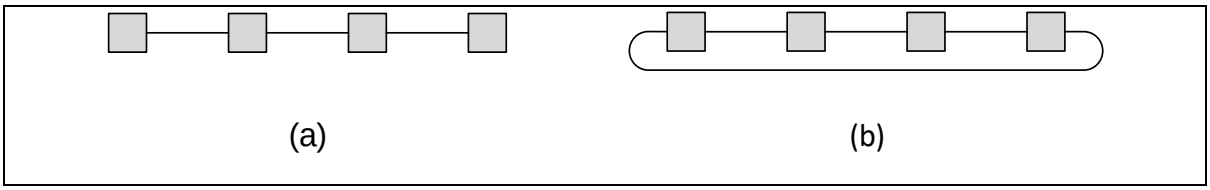


Şekil 7 Yıldız bağlantılı ağlar: (a) tamamen bağlı ağ ve (b) yıldız bağlı ağ [15].

2.4.1.6. Doğrusal Diziler, Örgü Ağlar ve Çok Boyutlu Örgüler

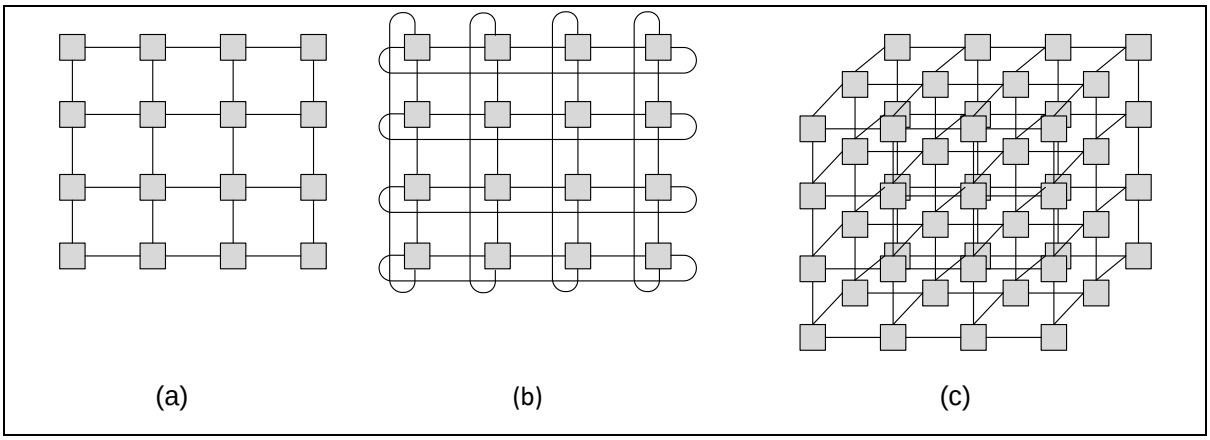
Doğrudan bağlı ağlardaki etkinliğin çok fazla bağlantı sebebiyle ölçeklendirilememesi ve yönetilememesi sebebiyle doğrusal diziler, daha sonrada örgü ağlar ve çok boyutlu ağlar oluşturulmuştur.

Doğrusal dizi her düğümünün sağında ve solunda olmak üzere iki komşu bağlantısı olduğu ağ topolojisidir. En baş ve sondaki düğümlerinde bağlanması ile halka ya da tek boyutlu torus ağı oluşturulmuş olur.



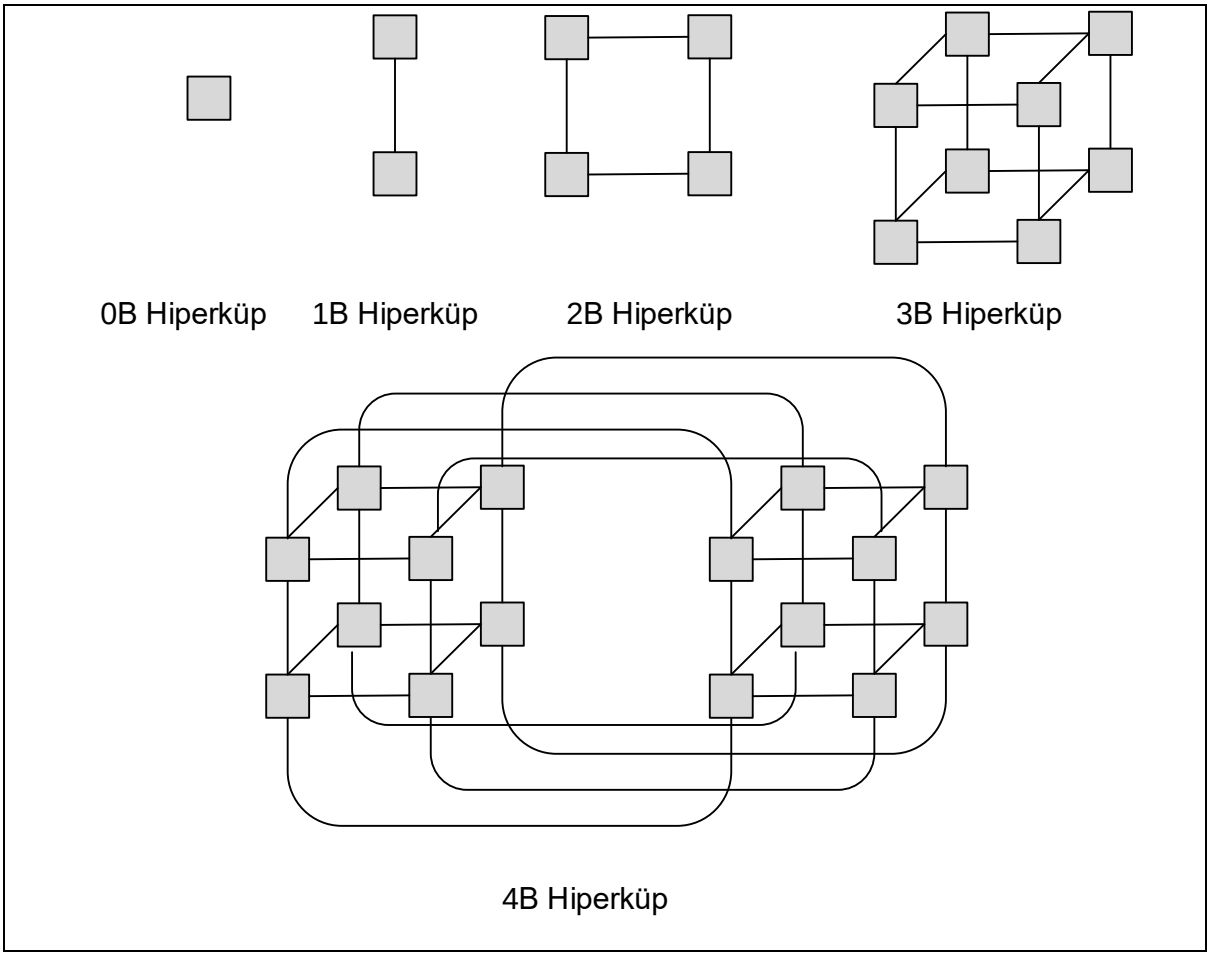
Şekil 8 Doğrusal diziler: (a) sarmal olmayan ve (b) sarmal bağlı [15].

Bu doğrusal dizilerin 2 boyutta yerleştirilmesi ile örgü ağlar gerçekleştirilir. 2 boyutlu örgü ya da torus ağlarda her düğümün dört adet bağlantısı bulunur. 2 boyutlu örgüler 2 boyutlu bir alanda serilerek kolaylıkla oluşturulabilir. Ayrıca hesaplamada 2 boyutlu veri yapıları sıklıkla kullanılması sebebiyle 2 boyutlu örgüler koşut sistemlerde en yaygın kullanılan topoloji olmuştur. Kenar noktalardaki düğümlerin de birbirleri ile bağlanması durumunda da 2 boyutlu torus ağları meydana gelir. 2 boyutlu örgülerin yan yana kullanılması ile de 3 boyutlu örgüler oluşturulur.



Şekil 9 İki ve üç boyutlu örgüler: (a) sarmal olmayan iki boyutlu örgü, (b) sarmal bağlı iki boyutlu örgü ve (c) sarmal olmayan üç boyutlu örgü [15].

Boyutların çoğaltılması sonucu çok boyutlu örgüler oluşturulur. Yukarıda bahsedilen doğrusal diziler ve örgü ağlar da çok boyutlu (k-d) örgü ağlar sınıfında olarak düşünülebilir. Bu sınıfa aynı zamanda hiper küp (*hypercube*) adı verilir. Bir k boyutlu hiper küp topolojisinde 2^k kadar düğüm bulunur.



Şekil 10 Hiper küplerin oluşturulması [15].

2.5. KOŞUT PROGRAMLAMA

Birçok işlemci ile geliştirilen koşut bilgisayar sistemlerinin oluşturulması, bu sistemler üzerinde çalışması istenen koşut yazılımların geliştirilmesini beraberinde getirir. Yüksek başarım gerektiren bilim ve mühendislik alanında hesaplama benzetim uygulamalarından veri madenciliği gibi pek çok alanın koşut işlem için uygulama alanı olmaktadır [15]. Bu alanlara kısaca göz atacak olursak:

Mühendislik ve tasarım uygulamaları: Hava araçlarının kanat tasarımı, içten yanmalı motor tasarımı, yüksek hızlı devreler, mikroelektromekanik ve nano teknoloji sistemlerin tasarımı gibi alanlarda yüksek başarımlı hesaplama kullanılmaktadır. Bu tasarım uygulamalarında koşut işlem ile kesikli ya da sürekli eniyileştirme problemleri çözülmektedir.

Bilimsel uygulamalar: Son yıllarda yüksek başarımlı programlama bilimsel uygulamalarda yaygın olarak kullanılmaya başlanmıştır. Özellikle biyoenformatik alanında genlerin ve proteinlerin işlevsel ve yapısal karakterleri üzerine benzetim sistemleri oluşturulma ihtiyaçları ortaya çıkmıştır. Bu biyolojik serilerin analizi ile ilaç tasarımı, hastalık tedavileri ve medikal durumların teşhisleri gibi alanlarda koşut işlem uygulanmaya başlamıştır. Fizik ve kimya alanlarında da kuantum olayları ve makro moleküler sistemlerin tanımlanması için kullanılmaktadır. Bu sayede yeni madde tasarımları, kimyasal yolların tanımlamaları yapılabilmektedir. Astrofizik alanında galaksilerin ve termonükleer süreçlerin benzetimleri yapılabilmektedir. Hava modellemesi, mineral görünümü ve akışkan mekanikleri de diğer uygulama alanlarındandır.

Ticari uygulamalar: Internet kullanımının yaygınlaşması ile ölçeklenebilir performans sunan maliyet etkin sunucu ihtiyaçları artmıştır. Ağ ve veri tabanı sunucularının bu maliyet etkinliği sağlaması için koşut işlem uygulamalarından yararlanılmaktadır. Ayrıca büyük ölçekli veriler içinde pazar kararlarının bulunması için veri madenciliği uygulamaları gibi alanlar da bulunmaktadır.

Bilgisayar bilimleri uygulamaları: Bilgisayar bilimlerinde güvenlik ve tehdit algılama uygulamaları önem kazanmaktadır. Örneğin tehdit algılamada, ağ üzerinde gelen verilerin toplanıp gerçek zamanlı olarak analiz edilerek tehditlerin

yakalanması gerekmektedir. Ayrıca şifrelemede daha zor kırılabilen şifrelemeler için yüksek başarımlı hesaplama yapmak gerekebilir. Bir diğer uygulama alanı da gerçek zamanlı sistemlerdir. Yüksek hızlarda gerçek zamanlı çalışan sistemlerin yüksek başarımlı hesaplama yapma ihtiyacı artmaktadır.

Tüm bu uygulama alanları günümüzde gittikçe önem kazanmakta ve yüksek başarımlı hesaplama yapabilmek amacı ile koşut işlem algoritmaları ve koşut sistemlerin kullanımı gerekmektedir.

2.6. KOŞUT ALGORİTMA TASARIMI

Problemlerin çözümünde algoritma geliştirme önemli bir rol oynar. Bir sıralı algoritma, sıralı bir bilgisayar kullanarak verilen bir problemin çözüm adımlarını içerir. Koşut bir algoritma da verilen problemin çoklu işlemcide çözümüdür. Ancak koşut algoritmanın tanımlanması sadece adımların tanımlanmasını içermez. En azından birlikte çalışılabilirlik durumlarını ve aynı anda çalışacak algoritma adımlarını içermek zorundadır. Bu çoklu işlemci kullanımının faydasının belirlenmesi için gerekir. Birlikte çalışacak işlerin neler olduğu, bu iş parçalarının hangi işlemcilere dağıtılması gerektiği, girdi/çıktı ve ara verilerin nasıl paylaşılması gerektiği, paylaşılan veriye erişimin yönetimi ve koşut algoritmanın yürütülmesinde çeşitli adımlarda işlemcilerin eşzamanlı hale getirilmesi de önem taşır.

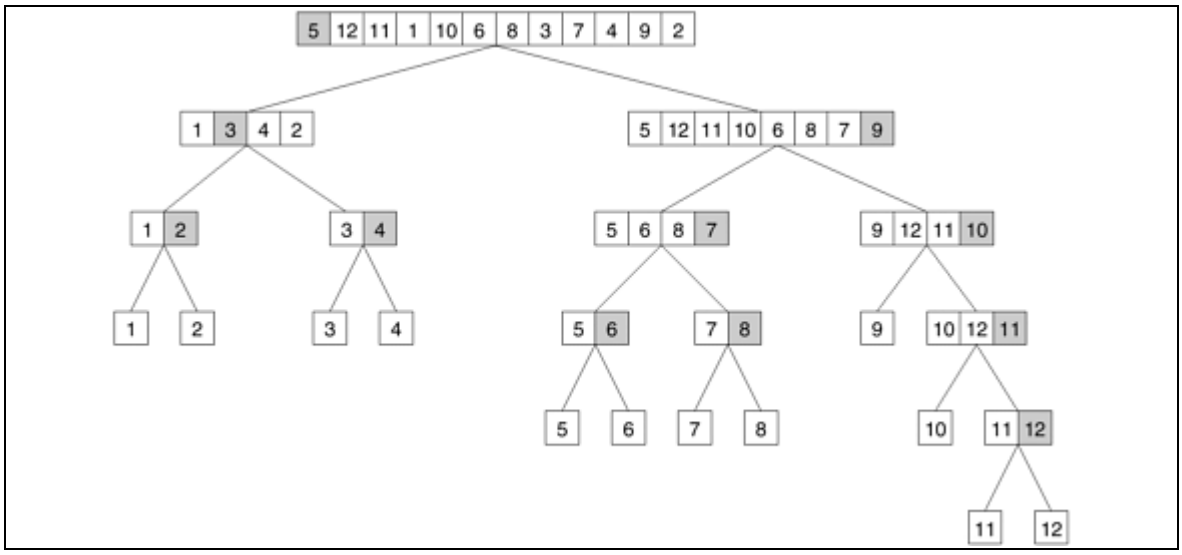
2.6.1. Parçalama Teknikleri

Koşut olarak bir problemin çözümü için gerçekleştirilecek hesaplamaların eşzamanlı yürütülmesini sağlamak üzere işlemler kümelerine parçalanması gerekir. Bunun için bazı parçalama teknikleri kullanılmaktadır. Bu teknikler genel olarak özyinelemeli (*recursive*) parçalama, verisel (*data*) parçalaması, keşifsel (*exploratory*) parçalama ve fırsatçı (*speculative*) parçalama olarak sınıflandırılabilir. Burada özyinelemeli ve verisel parçalama teknikleri genel amaçlı ve birçok problemin çözümünde kullanılabilen tekniklerdir. keşifsel ve fırsatçı parçalama teknikleri ise problemlerin doğasına göre oluşturulmuş parçalama tekniklerindendir.

2.6.1.1. Özyinelemeli Parçalama

Özyinelemeli parçalama problemlerin parçala ve feth et (*divide-and-conquer*) yöntemi ile çözülebilen problemlerinde kullanılmaktadır. Her problem daha küçük alt

problemlere ayrılır ve ayrılan her alt problem de özyineli olarak daha küçük problemlere parçalanır.



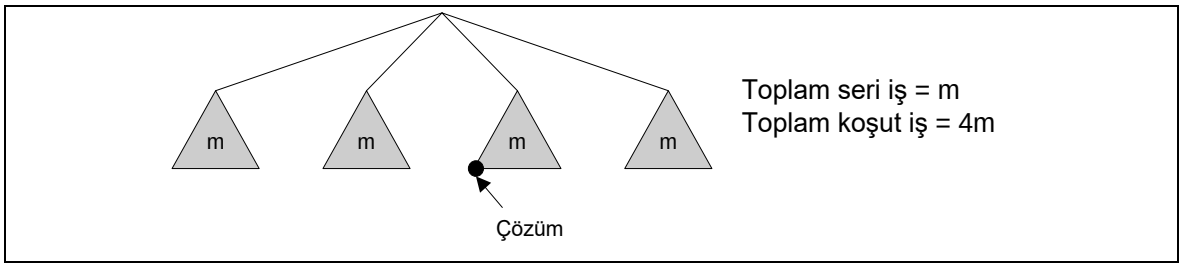
Şekil 11 Quicksort sıralama algoritmasının özyineli parçalama ile görev atamalarının belirlenmesi.

2.6.1.2. Verisel Parçalama

Verisel parçalama büyük veri kümeleri üzerinde çalışan algoritmaların koşut olarak tasarlanmasında sık kullanılan ve güçlü bir yöntemdir. Büyük veri kümeleri koşut sistemde bulunan işlemcilere dağıtılır ve her bir işlemci kendine dağıtılmış olan veri üzerinde işlem yapar. Dağıtılan veriler üzerinde yapılan hesaplama genelde her işlemcide aynıdır. Sonuçlar toplanarak daha küçük bir set üzerinde hesaplanır. Verilerin bölümlenmesi ve dağıtılmasında da çeşitli teknikler kullanılmaktadır. Genelde kullanılan tek bir kaynağın veriyi parçalayıp sistemdeki işlemcilere dağıtmasıdır.

2.6.1.3. Keşifsel Parçalama

Bu yöntem problemin bir çözüm aralığının aranmasına bağlı hesaplamaları içerdiği durumlarda kullanılabilir. Örneğin bir ağaç üzerinde derinlik önce arama yapılmasında ağacın çeşitli durumlarının işlemcilere dağıtılarak her işlemcinin kendi ağacı üzerinde derinlik önce arama yapması bu yöntem ile çözülebilir.



Şekil 12 Keşifsel olarak her ağaç aramasının ayrı işlemcilere parçalanması

2.6.1.4. Fırsatçı Parçalama

Fırsatçı parçalama bir program daha önce yapılmış olan hesaplamalarda oluşan çıktılara bağlı olarak birçok durumdan herhangi birini seçmesi gerektiğinde kullanılan bir yöntemdir. Bu durumda bir görev bir sonraki aşamada kullanılacak olan çıktıyı üretirken, diğer görevler bir sonraki aşamada yapılacak hesaplamayı gerçekleştirirler. İlgili çıktı üretildiğinde diğer görevlerden uygun olan hesaplamanın sonucu kullanılırken diğer görevlerin sonuçları atılır. Atıl hesaplamalar gerçekleştirilmiş olsa da toplamdaki hesaplama süresi kısaltılmış ve algoritmanın etkinliği arttırılmış olur.

2.6.1.5. Hibrit Parçalama

Verilen bu parçalama yöntemleri diğer yöntemler ile kullanılmaya uygundur. Çok sık olarak kullanılan ve bu tez kapsamında da gerçekleştirilen yöntemin temelini oluşturan özyinelemeli ve verisel parçalama yönteminin birlikte kullanımı genel amaçlı problemlerin çözümünde uygun bir yöntemdir. Büyük verilerin işlemcilere dağıtılması amaçlanırken bu verilerin dağıtılmasında özyineleme kullanılarak etkinlik arttırılabilir.

2.6.2. Dağıtım Teknikleri

Bir hesaplama alt görevlere parçalandıktan sonra bu görevlerin hangi işlemler üzerinde gerçekleştirileceğinin adreslenmesi gerekmektedir. Yürütme zamanını azaltmaya çalışırken de koşut çalışmak için gereken ek yüklerin en aza indirilmesi gerekir. Yürütülecek görevlerin parçalanması sonucunda iki önemli ek yük ortaya çıkar: işlemler arası haberleşme sırasında harcanan zaman ve boşta bekleyen işlemcinin harcadığı zaman. Adreslemenin iyi yapıldığı algoritma tasarımları bu iki ek yükü en aza indirmeyi sağlamalıdır.

2.7. AŞIRI ÖLÇEKLİ KOŞUT SİSTEMLER

Aşırı ölçekli (*exascale*) koşut sistemler, her gün büyüyen ve daha çok işlem kapasitesine sahip olan koşut sistemler için kullanılmaya başlayan bir tanımlamadır. 2015 yılı itibari ile 10 yıl içinde işlem kapasitesinin 1000 katına çıkması beklenmektedir [2]. İlerlemekte olan teknolojik gelişmeler, gelecekte çok büyük işlem kapasitesine sahip bilgisayarların oluşturulacağına işaret etmektedir. Ancak bu ilerleme, çözülmesi gereken bazı büyük problemleri de beraberinde getirmektedir.

Bu büyük problemler dört ana sınıfa ayrılabilir:

- Enerji ve Güç Problemi: Daha güçlü ve daha çok işlem kapasitesi olan sistemlerin yapılması beraberinde daha çok güç tüketimi ihtiyacı ortaya çıkarmaktadır. Bu güç tüketimini karşılayacak enerjinin sağlanması oldukça önemli bir problemdir. Bu noktada daha az güç sarfiyatının sağlanmasına yönelik çalışmalar yapılması gerekmektedir.

- Bellek ve Depolama Problemi: Daha yüksek işlem kabiliyeti daha büyük veriler üzerinde çalışılması ile anlamlı olmaktadır. Daha büyük verilerin de bellekte ve depolama aygıtlarında ne şekilde saklanması ve erişimlerinin ne şekilde yapılması önemli bir sorundur.

- Eş zamanlılık ve Yerellik Problemi: İşlem kapasitesinin artması ile sistem performansının yükseltilmesi için işlemlerin eş zamanlı olarak çalıştırılabilmesi ve yerel olarak işlemlerin yapılarak sonuçların toplanabilmesi gerekmektedir. Bu durum koşut programlama problemlerini ön plana çıkarır. Seri programlamadan farklı olarak koşut algoritma tasarımlarının aşırı ölçekli koşut sistemlere göre yapılması gerekliliğini çıkarır.

- Esneklik Problemi: Sistemin hata durumlarında ya da performans değişimlerinde de normal çalışma durumunda olduğu gibi doğru çalışmasının sağlanması gerekmektedir.

Bu problem göz önüne alındığında aşırı ölçekli sistemler için geliştirilmesi gereken çalışma alanları şu şekilde sıralanabilir:

1. **Aşırı ölçekli donanım teknolojileri ve mimarilerinin geliştirilmesi ve eniyileştirilmesi.** Bu alanda haberleşme, bellek ve işlem için kullanılan donanımların enerji etkin biçimde geliştirilmesi, alternatif düşük enerjili cihazların tasarlanması gibi konular sayılabilir.

2. **Aşırı ölçekli yazılım mimarilerin ve programlama modellerinin geliştirilmesi ve eniyileştirilmesi.** Yeni donanım mimarilerin geliştirilmesi ile eş zamanlılığın aşırı derecede artması sonucunda programlama modellerinin de bu donanımları etkin biçimde kullanması beklenmektedir. Bu alanda yerelden gelen ve haberleşme etkin mimarilerin geliştirilmesi, veri dağıtımında enerji tüketiminin azaltılması ve milyar-düzeyde iş parçacığını idare edebilecek mimari ve programlama modelinin geliştirilmesi sayılabilir.

3. **Aşırı ölçekli algoritmalar, uygulamalar, araçların geliştirilmesi.** Geliştirilen yeni mimariler ve yeni teknolojiler ile uyumlu alternatif algoritmalar, uygulamalar ve araçların geliştirilmesi gerekecektir.

4. **Esnek aşırı ölçekli sistemlerin geliştirilmesi.** Donanım ve algoritma karmaşıklığı ile ilgili problemler ve yeni aksaklığa dayanıklılık mekanizmalarının üzerinde çalışmalar gerekecektir.

Bu çalışma alanları incelendiğinde bu maddelerin disiplinler arası çalışmalar olması gerektiği görülmektedir. Örnek olarak güç tüketiminde daha az güç tüketen devrelerin tasarlanması gerekirken aynı zamanda o devrelere ait bileşenlerin de az tüketen yapıda oluşturulması gerekecektir. İşlem birimlerinin de güç tüketimi için haberleşme maliyetlerinin eniyileştirilmesi gerekirken, bunu sağlayacak yazılım mimarilerinin de gerçekleştirilmesi gerekecektir.

Bu noktada yazılım alanında aşırı ölçeğe geçildiğinde ortaya çıkan önemli sorunlar belirmektedir [16]. Bu sorunlar incelendiğinde en önemli sorunlar şu şekilde sıralanabilir:

- Şuan hali hazırdaki yazılım yaklaşımları, gelecekteki Büyük Zorluklu (*Grand Challenge*) uygulamaların aşırı ölçekli sistemlerde kullanımında yetersiz kalacaktır.

- Çakışma (*contention*), kritik kaynakların eş zamanlı istekler sebebiyle engellenmesini azaltmak ya da hafifletmek için önemli bir etmendir.

- Başarılı bir aşırı ölçekli çalışma modeli, eşzamanlılık, yerellik, enerji tüketimi ve ek yük içeren gelecek aşırı ölçekli sistemlerde performans ölçütlerini gerçekleştirmek zorundadır.

- Uygulamaları büyük sayıda çekirdeklere ölçeklemek, iki bölümlü bant genişliğinden (*bisection bandwidth*) tam olarak yararlanılmasını gerektirir. Bant genişliği gereksinimi mimari olarak darboğaz oluşturmamalıdır.

Bu kapsamda düşünüldüğünde aşırı ölçekli koşut sistemlerin gelişmesi için hem genel sistem mimarileri anlamında, hem de yazılım alanına özgü olarak çeşitli problemlerin giderilmesi gerekmektedir. Aşırı ölçekli koşut sistemlerin hayata geçirilmesi çok uzak bir gelecek olmamasına rağmen bu problemler hala var olmaya devam etmektedir. Koşut sistemlerin geleceğinin de aşırı ölçekli biçimde devam edeceği görülmektedir.

3. MODEL GÜDÜMLÜ YAZILIM GELİŞTİRME

3.1. GİRİŞ

Yazılım geliştirme pek çok kişi tarafından kolay gibi gözükmesine rağmen aslında işin içine girdikçe pek çok problem ile karşılaşılır. Öncelikle problem alanının geliştirici tarafından tam olarak anlaşılması çok önemlidir. Bu noktada alan uzmanı kişilerden yardım alınır ancak bu kişilerin de yazılım geliştirme konusundaki bilgisi sınırlıdır [17]. Pratikte de bu durum oldukça vakit alan bir öğrenme sürecine dönüşür. Bunun yanında yazılım geliştirmenin doğasında geliştirme için gereken zaman ve işçilik, diğer sistemler ile birlikte çalışması, takım çalışması içinde geliştirilen farklı bileşenlerin bütünleşmesi gibi kısıtlar her zaman göz önüne alınmalıdır. Her geliştirme safhasında yapılan değişiklikler gözden geçirilmeli ve bitti diyebilmek için geliştirme sürecindeki her takım üyesinin farklı açılardan yazılım geliştirme sürecine katkıda bulunması ve bu süreçlerin yönetilmesi gerekmektedir.

Yazılım mühendisleri tüm bu sorunları çözebilmek için çeşitli yöntemlere başvurmaktadır. Bu yöntemlerden en etkilisi ise modellemedir. Modeller gerçek sistemlerin bir soyutlamasıdır. Bu sayede mühendislerin çok detaya girmeden ana noktalara odaklanması sağlanabilir. Farklı paydaşlar tarafından farklı bakış açıları ile değerlendirilen modeller, geliştirme tamamlanmadan bir fikir ortaklığı sağlar ve büyük sorunların önceden fark edilmesini sağlar.

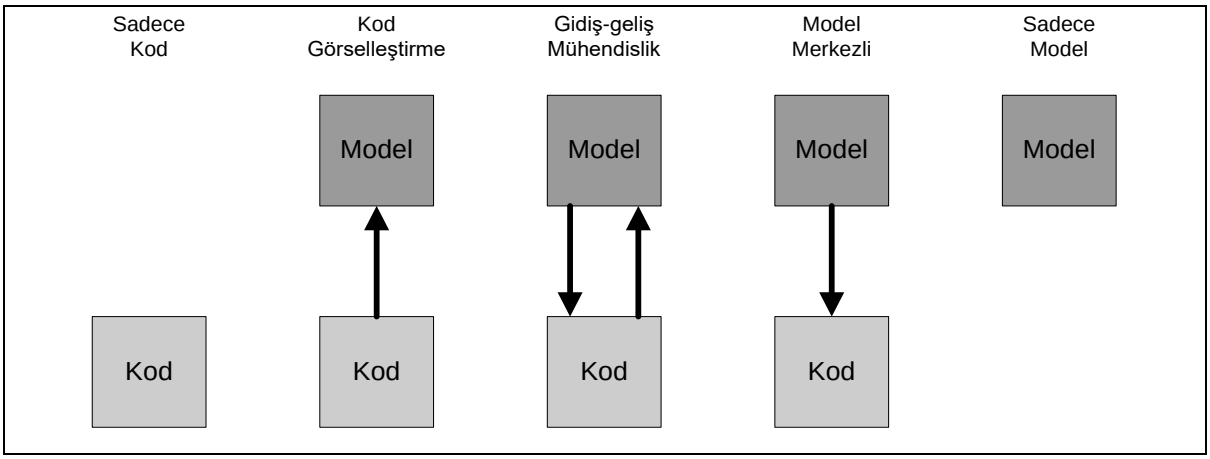
Yazılım geliştirme yaklaşımları değerlendirildiğinde de modeller, modelleme ve model dönüşümleri model-güdümlü geliştirme fikrini ortaya çıkarmaktadır. Modeller problem alanında ve çözüm alanında kullanılabilir. Model içeriklerinin çözümlenmesi, bu içerikler arasındaki bağlantıların belirlenmesi ve modelden-modele dönüşüm model-güdümlü geliştirmenin temelini oluşturmaktadır.

Model güdümlü geliştirme için örnek olarak model-güdümlü mimari (*model-driven architecture*), modele tümleşik hesaplama (*model-integrated computing*) ve yazılım fabrikası (*software factory*) verilebilir [18][19]. Model-güdümlü mimari [20] Object Management Group (OMG) tarafından öngörülen bir yazılım geliştirme yöntemi olarak sunulurken, yazılım fabrikaları da ürün hattı kapsamında ürün geliştirme süreci otomasyonu sağlayan bir yöntem olarak görülebilir [21].

3.2. MODELLEME

Modelleme yazılım geliştirme sürecinde ilk safhalardan itibaren yazılım mimarlarına ve geliştiricilerine yardımcı olan bir kavramdır. Sistemlerin bir soyutlaması olarak belirtilen modellerin oluşturulması için de bazı araç ve gösterimler belirlenmeye çalışılmıştır. Bu çalışmaların ilk örneği ise Birleşmiş Modelleme Dili (*Unified Modelling Language* - UML) olmuştur [22]. UML geliştiricilere pek çok sistem karakteristiğinin modelleyebilecekleri bir gösterim sunar.

UML gibi modelleme gösterimlerinin pratikte kaynak kod ile nasıl eşleştiği önemli bir özelliktir. Yazılım geliştirmede modelleme yaklaşımları bu eşleştirmeye bağlı olarak değişir. Şekil 13, yazılım mühendisleri tarafından son ürün olan kaynak kodun model ile nasıl ilişkilendirildiğini göstermektedir [17].



Şekil 13 Modelleme kod ilişkileri.

Günümüzde hala sadece kaynak kod (*code only*) yazarak geliştirilen uygulamalar mevcuttur. Bu tarz yazılım geliştirmede soyutlama kod düzeyinde paketler modüller ve arayüzler seviyesinde olmaktadır. Mimari tasarım genelde geliştiricilerin kafasında, çalışma materyallerinde ya da sunum materyallerinde mevcuttur. Küçük ölçekli yazılımlarda ve küçük yazılım ekiplerinde etkili olmasına karşın, karmaşıklığın artması durumunda sistemin önemli noktaları gözden kaçabilmektedir ve bir süre sonra yönetimi zor bir hal almaktadır.

Kod görselleştirme (*code visualization*) geliştiricilere geliştirdikleri kod ile ilgili ekstra bir görüş açısı kazandırır. Grafiksel bir gösterim ile görselleştirilen kod sayesinde geliştiriciler görsel olarak kodların yapısını inceleyebilirler. Hatta bazı araçlar görsel

olarak kaynak kodun değiştirilmesine olanak vermektedir. Bu yapıya kod modeli de denmektedir.

Gidiş-geliş mühendislikte (*roundtrip engineering*) model ile kod sürekli olarak senkronize tutulmaya çalışılır. Geliştirici bir kısım detayı model içinde belirler. Daha sonra modelden koda dönüşüm ile kodu oluşturur. Daha ayrıntılı detaylar kod içinde yazılır ve modele eşlenir. Bu işlem geliştirme aşamasında birkaç kez tekrar edilir.

Model merkezli (*model-centric*) yaklaşımda geliştirici tamamen model üzerinde çalışır. Kodun oluşturulması modelden koda dönüşüm ile gerçekleştirilir. Kod oluşturma için çeşitli örüntüler ve yazılım çatıları kullanılır. Model burada asıl çıktıdır ve geliştirici bu model üzerinde çalışır.

Bir kaynak kod ihtiyacı olmayan ve genellikle iş seviyesinde (*businness-level*) kullanılan sadece model (*model only*) yaklaşımında, geliştirici sadece model tasarımı yapar. Genellikle işin yapısını tartışmak ve anlamak için yapılan bir geliştirme yöntemidir.

3.3. MODEL GÜDÜMLÜ GELİŞTİRME

Modelleme yukarıda verilen beş ayrı yaklaşım ile sınıflandırılabilir. Bu modelleme yaklaşımlarında model ve kaynak kodun birbirini yansıtmasının sağlanması için araç desteklerinin önemi büyüktür. Model güdümlü geliştirme yaklaşımları da bu noktada devreye girmektedir. Yazılım geliştirme faaliyetlerinin hızlandırılması için kullanılan Yazılım Ürün Hattı Mühendisliği (*Software Product Line Engineering*), bir uygulama alanı için yazılım geliştirme faaliyetlerini ortak karakteristiklere uygun olarak sistematik biçimde eniyilemeyi amaçlar [18]. Model Güdümlü Yazılım Geliştirme (*Model Driven Software Development* - MDSD) ürün hattında bu amaçla kullanılan en önemli yöntemlerdendir.

Yazılım mühendisliği alanında önemli bir yere sahip olan dönüşüm, model dönüşümü göz önüne alındığında belli kurallara ve örüntülere göre yapılmaktadır. Bu dönüşümdeki kuralların belli bir alana özgü olması sebebi ile verilen kurallara bağlı olarak geliştirilen bir dil meydana gelmektedir. Bu dile alana özgü dil (*domain-specific language*) denir. Alana özgü dil, genel amaçlı dillerden farklı olarak belirli bir işlem kümesi için kullanılmak üzere tanımlanmış dildir [23].

Model güdümlü geliştirmede oluşturulan modelden kaynak kodun üretilmesi dönüştürme (*transformation*) ile sağlanmaktadır. Ayrıca çeşitli detay seviyelerine göre farklı seviyelerde modellerin de oluşturulması gerekebilir. Bu da bir modelden başka bir modele dönüşüm ile gerçekleştirilebilir. Model güdümlü geliştirmenin ana adımlarından biri kabul edilen dönüştürme işlemleri temelde iki sınıfa ayrılmaktadır:

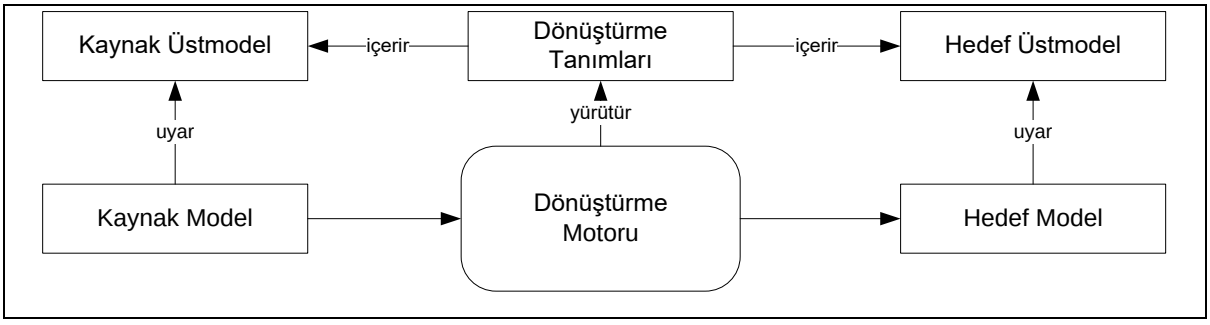
- Modelden modele dönüşüm
- Modelden metne dönüşüm

3.3.1. Modelden Modele Dönüşüm

Modelden modele dönüşüm, bir modelden başka bir model ya da model kümesine gereken bilgilerin donüştürülmesi işlemini kapsar. Bu dönüşüme en güzel örnek sınıf mimarisi oluşturulmuş bir modelden veritabanı şemasına ya da XML biçimındeki adresleme kütüklerine dönüştürme verilebilir. Farklı detay seviyelerinde tanımlanması istenen modellerin arasındaki dönüşümler de bu sınıfa girmektedir.

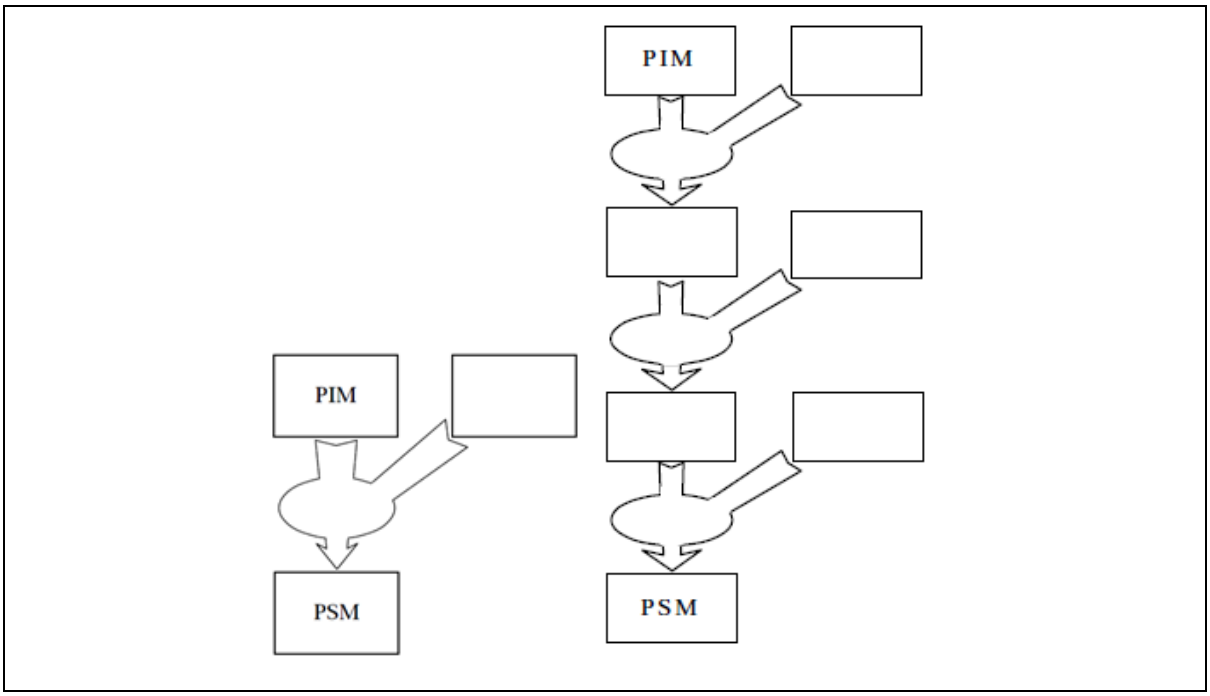
Modelden modele dönüşümde alana özgü dilin tanımlanması önemlidir. Alana özgü bir dil için sözdizim (*syntax*) kullanılması gerekmektedir. Ecore [24], Eclipse projesi kapsamında MOF (*Metaobject Facility*) [25] tanımlamasına uygun olarak geliştirilen bir alana özgü dil sözdizimidir. EMF (*Eclipse Modelling Framework*)[24] Ecore için geliştirme araçları sunmaktadır, bu sayede alana özgü dil oluşturulması sağlanabilmektedir.

Ecore kullanılarak tanımlanan üst model (*metamodel*), model güdümlü geliştirilecek yazılımın modelleme gösteriminin soyut sözdizimini verir [19]. Bu üst model kullanılarak gerçekleştirilecek olan model dönüşüm temel kavramı Şekil 14'de verilmiştir.



Şekil 14 Model dönüşümü kavramsal görünümü.

Modelden modele dönüşümün sağladığı en önemli özelliklerden biri oluşturulan modelin platformdan (örneğin donanımdan, uygulama çatısından ya da programlama dilinden) bağımsız olmasını sağlamaktır. Bu amaçla bir modelden belirli seviyelerde farklı bilgiler içeren modellere dönüşüm yapılır. Model güdümlü mimaride (*model driven architecture* - MDA) bu modeller platformdan bağımsız model (*platform independent model* - PIM), platforma özgü model (*platform specific model* - PSM) ve platform modeli (PM) olarak belirtilmiştir. Model dönüşümü de Şekil 15'de olduğu gibi verilmektedir. Model dönüşümü birkaç aşamada da kullanılabilir.



Şekil 15 Model dönüşümü.

Bu aşamada dönüşümde kullanılan kuralların tanımları önem kazanır. Kaynak modelin hedeflenen modele dönüşümde kuralların ne şekilde uygulanacağının belirlenmesi için hem kaynak model hakkında, hem de hedef model hakkında bilgiye ihtiyaç vardır. Bu bilgi yine bir model sözdiziminde verilebilir. Model hakkında bilgiyi içeren modele üst model (*metamodel*) denir. Dönüşüm tanımlamaları kaynak üst modeli ile hedef üst modeli arasında tanımlanır. Bu durumda bir dönüşüm motoru (*transformation engine*) kullanarak bu tanımlamalara uygun biçimde kaynak modelden hedef modele dönüşüm gerçekleştirilir.

Burada kaynak modelin kaynak üst modele bağımlılığını görebiliriz. Kaynak üst model yazılım geliştiricisi tarafından oluşturulacak modelin sözdizimini belirlediği

için bir dil vazifesi görür. Kullanılacak olan alan (*domain*) için tanımlandığı için üst model bir alana özgü dil olmaktadır.

3.3.2. Modelden Metne Dönüşüm

Modelden metne dönüşüm, model ya da model kümesindeki bileşenlerin metin kesimlerine dönüştürülmesine dönüştürülmesi olarak tanımlanabilir. Bu dönüşüm genelde kod oluşturma amaçlı kullanılmasına rağmen sadece programlama dilleri ile sınırlı kalmaz. Programlama dilleri dışında yapılanma kütüklerinin oluşturulması, ileti şemalarının oluşturulması, veri tanımlamalarının oluşturulması gibi amaçlarla da kullanılabilir.

Genelde önem olarak modelden modele dönüşümün gerisinde düşünülse de yazılım geliştirme sürecinin en önemli çıktısı olan kaynak kodun üretilmesi için modelden metne dönüşüm kaçınılmazdır. Modelden metne dönüşümde genelde şablonlar kullanılmaktadır. Xpand [26] gibi kanıtlanmış bir şablon dili ile Eclipse üzerinde geliştirilen araçlarla modelden metne dönüşüm uygulanmaktadır.

Kod üretimi sadece modele özgü metinlerin genel arama karakterleri kullanılarak dönüştürülmesini içermez. Bu dönüşümde de modelden metne dönüşüme uygun alana özgü dil tanımları yapılır. Ancak bu alana özgü dil, şablonlar kullanılarak tanımlanır.

4. KOŞUT İŞLEMİN MODELLENMESİ

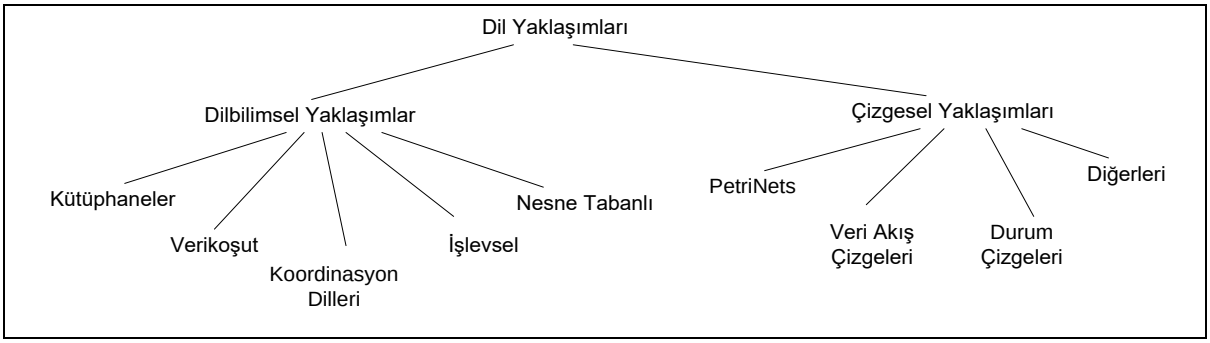
Uygulamaların geliştirilmesinde, özellikle de koşut işlem içeren uygulamaların geliştirilmesinde programlama modeli oldukça önemlidir. Koşut işlemde örnek olarak ileti geçirme (*message passing*) çok sık kullanılan bir yöntemdir. Bazı uygulamaların geliştirilmesinde bu yöntem basit bir model sunar. Ancak bazı uygulamalarda bu kullanım çok alt düzey ve esnek olmayan bir model sunar. Bu durum da programların karmaşıklığını arttırır ve algoritma tasarım kararlarını kısıtlar.

Düşük seviye programlama modellerinin kullanılması yerine üst seviye modellerin kullanılması, prototipleme gibi yöntemlere olanak verirken, tasarım kararlarının daha kolay uygulanmasını sağlamaktadır. Üst seviye programlama modelleri de alan uzmanlarına daha kolay ve hızlı tasarım yapma imkanı kazandırmaktadır.

Bir programlama modeli bir hesaplama dili (*computational language*) ve bir eşgüdüm dili (*coordination language*) olmak üzere ikiye ayrılabilir [27]. Bu iki dil birlikte tam bir programlama sistemini meydana getirebilir. Bu iki dil aslında model-güdümlü yazılım geliştirme yaklaşımında bulunan platformdan bağımsız üst model ile platforma özgü üst model arasındaki ilişki ile aynıdır. Bu üst modeller dilin sözdizimini meydana getirirken platformdan bağımsız modeller daha üst düzey olan eşgüdüm dilini karşılamakta, platforma özgü modeller ise kaynak koda dönüştürülebilir olan hesaplama diline karşılık gelmektedir.

Gelernter ve Carriero [27] bir programlama modelinin iki ayrı dilden oluşturulmasının daha iyi bir seçenek olduğunu belirtmektedir. Bir hesaplama dili, eşgüdüm dili olmadan kullanılamaz. Hesap yapmak, o hesaplamanın diğer hesaplamalar ile birlikte anlamlandırılması sonucunda kullanılabilir hale gelir.

Hasselbring'in oluşturduğu sınıflandırmada da bu tip eşgüdüm dilleri koşut işlem göz önüne alınarak sınıflandırılmaktadır (Şekil 16) [28].



Şekil 16 Koşut dillerin sınıflandırılması.

Hasselbring'in [28] sınıflandırması ele alındığında diller,

- Dilbilimsel yaklaşımlar
- Çizgesel yaklaşımlar

olarak ikiye ayrılmaktadır. Dilbilimsel diller alana özgü kütüphaneler, kümesel veri koşutluğu, eşgüdüm dilleri, işlevsel diller ve nesneye yönelik diller olarak ayrılmaktadır. Çizgesel yaklaşımlar ise Petri-Ağları, veri akışı çizgeleri, durum geçiş çizgeleri ve diğer görsel diller olarak ayrılmıştır.

4.1. KOŞUT İŞLEM MODELİ

Model-güdümlü yazılım geliştirme yaklaşımının kullanımında oluşturulacak olan koşut işlem için modellemenin için ayrıntılara girmeden önce bu yaklaşımın bize ne kazandıracağını listeleyebiliriz:

- **Ölçeklenebilirlik:** Öne sürülen bu yaklaşımda kullanıcı, uygulamaya yönelik iletişim örüntülerini ve bu örüntülerin kullanılacağı döşemeleri (*tile*) tanımlayabilecektir. Koşut sistemin davranışı, algoritmayı oluşturan iletişim örüntülerinin bir dizisi (*sequence*) olarak belirlenir. Dizinin tanımlamasında olduğu üzere, iletişim örüntüleri ve döşemeler ağ boyutunun büyüklüğü ve işlem birimlerinin sayısı oranında özyineli ve/veya tekrarlayıcı biçimde büyütülebilir.
- **İki bölümlü bant genişliğinin etkin biçimde kullanılması:** Bir iletişim örüntüsü ve onun birlikte kullanıldığı temel döşeme, iki bölümlü bant genişliğinin ne etkinlikte kullanıldığını gösterir. Bu küçük ve temel döşeme üzerinde iletişim örüntüsünün birbiri ile çakışmayan etkin iletişim yolları kullanılmasını kolaylaştırır. Bu çakışmaların önlenmesi bu döşeme üzerindeki küçük ağ yapısındaki iki bölümlü bant genişliğinin etkin olarak tasarlanmasını sağlar. Ölçekleme ile iletişim örüntüsü ve döşeme büyütüldüğünde de bu etkinlik aynen korunacaktır.

- **Çakışmadan bağımsız iletişim:** Temel örüntü üzerinde oluşturulan çakışmadan bağımsız iletişimler, özyineli ve/veya tekrarlayıcı biçimde ölçeklendiğinde yine çakışmadan bağımsız olacaktır.

- **Kilitlenmeden bağımsız iletişim:** Temel örüntü üzerinde oluşturulan kilitlenmeden bağımsız iletişimler, özyineli ve/veya tekrarlayıcı biçimde ölçeklendiğinde yine kilitlenmeden bağımsız olacaktır.

- **Yerellik:** Temel döşemeler göz önüne alındığında, bu döşemelerden ölçeklenen büyük döşemeler alt bölümleri açısından sıradüzensel bir kapsamda yerellik içerir. Yerel olarak tanımlanan iletişim örüntüleri büyük örüntülere de ölçeklenir.

- **Biçimsellik:** Biçimsellik sayesinde koşut kodda geçici ya da beklenmeyen iletişim isteklerinin eklenmesi engellenir. İletişim örüntüleri, döşemeler ve bu öğelerin özyineli ve tekrarlayıcı biçimde ölçeklenmesi koşut sistemin koşut davranışını kullanıcı açısından daha anlaşılır kılar. Bu sayede biçimsel yolla hataların önüne geçilir.

- **İşbirliği:** Koşutluk ile alana özgü kavramların ayrılması ile farklı disiplinlerdeki insanların birlikte çalışması sağlanır. Alan uzmanları koşutluğun sağlanması ile ilgili kısımlarla uğraşmayıp, sadece yapılacak koşut işlemin içeriğine yoğunlaşabilir.

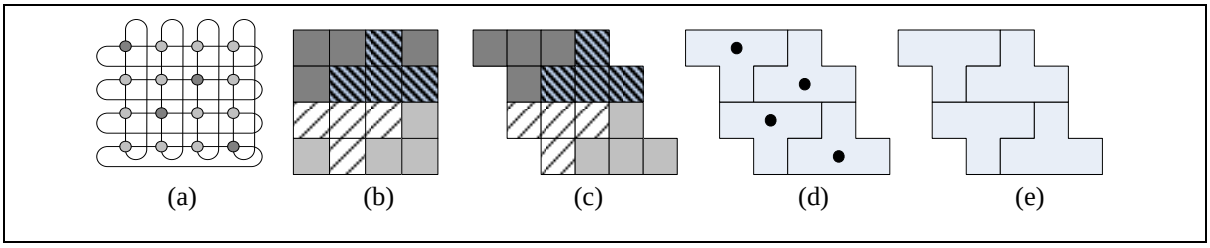
- **Yeniden kullanılabilirlik:** Temel döşemeler ve iletişim örüntüleri farklı uygulamalar ve algoritmalarda yeniden kullanılabilir.

- **Donanım ile eşgüdüm (donanım ve yazılımın birlikte tasarımı):** İletişim örüntülerinin belirlenmesi ile donanım üzerinde en az sayıda iletişim yolunun belirlenmesi ve donanım üzerinde bu iletişim yollarına göre donanımsal iyileştirmelerin yapılması sağlanabilir. Bu sayede iletişim gecikmeleri azaltılırken, aynı zamanda kullanılmayan iletişim yolları kapatılarak enerji tüketimi azaltılabilir.

Bu kapsamda koşut işlemin modellenmesi sırasında kullanılacak kavramların tanımları aşağıda verilmektedir.

4.1.1. Döşemeler

Döşemeler koşut modele ait temel yapı blokları olarak tanımlanabilir. Bir uygulamanın işlemlerinin işlemcilere dağıtılması ve bu işlemler arasındaki iletişim yollarının kurulmuş olan ağ altyapısı üzerinde yerleştirilmesinde temel döşemeler, birleşik döşemeler ve bu döşemelerin özellikleri önemli rol oynayacaktır. Bu döşemeler ve iletişim örüntüleri yeni algoritmaların oluşturulmasında yardımcı olacaktır [29].

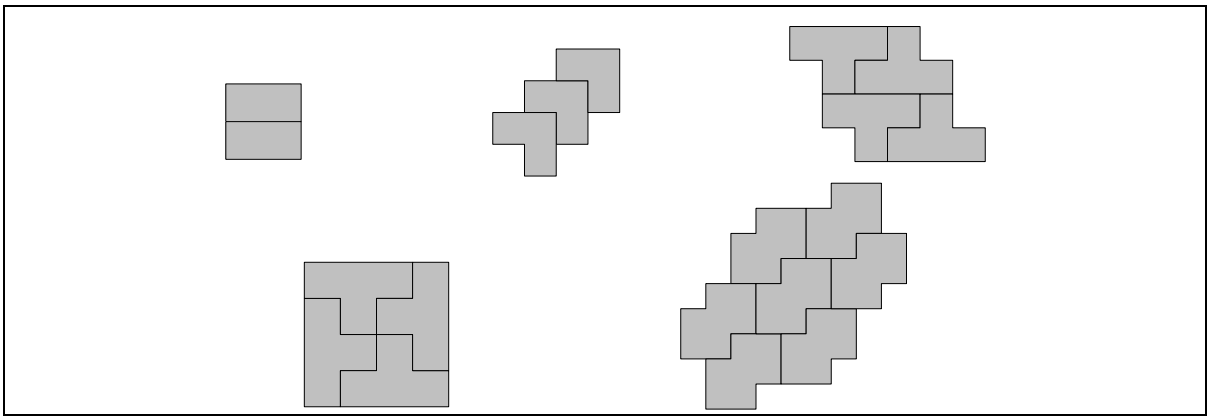


Şekil 17 4x4 torus topolojisinin farklı gösterimleri: (a) Geleneksel gösterim, (b) gölgeli kare gösterim, (c) açık kare biçimi, (d) hakim düğümler ile soyut gösterim ve (e) hakim düğümler olmadan soyut gösterim.

Şekil 17'de 4x4 bir torus bağlantılı koşut işlemcilerin gösterim biçimleri gösterilmektedir. 4x4 torus daha önce anlatıldığı üzere örgü şeklinde yerleştirilmiş işlemcilerin alt-üst ve sağ-sol uçlarının birleştirilmesi ile oluştrulmaktadır (Şekil 17 (a)). Şekil 17 (b)'de ise bu topolojinin döşemelere ayrıldığında ne şekilde ayrılabileceği gösterilmiştir. Bu aslında döşemeler daha düzgün biçimde ifade edilen Şekil 17 (c) ile aynı gösterimdir. Şekil 17 (d) ve (e) ise sadece döşeme biçimleri ile ve o döşemelere ait hakim (dominating) düğümlerin gösterildiği şekillerdir. Genelde NxN gibi bir torus gösteriminde N adet hakim düğümün olması esastır. Şekil 17 (d) ile belirtilen gösterimde her satır ve sütun boyunca bir adet olmak üzere toplam 4 adet döşeme ve hakim düğüm mevcuttur.

Döşemenin şekli kullanılan topolojiye ve tasarlanan algoritma ya da programın özelliğine göre değişebilir. Asıl amaç algoritmada kullanacağımız topolojinin tümünü kaplayabilmektir. Örnek döşeme şekilleri Şekil 18'de verilmiştir.

Döşemeler aslında kullanıcıların iletişim örüntülerini önceden tanımlı döşemeler kullanarak biçimsel olarak oluşturmalarını sağlar. Bu biçimsellik, geçici ya da beklenmeyen iletişimlerin önüne geçilmesini sağlar ve algoritma yapısını biçimsel biçimde geçerliliğini denetleyebilir.

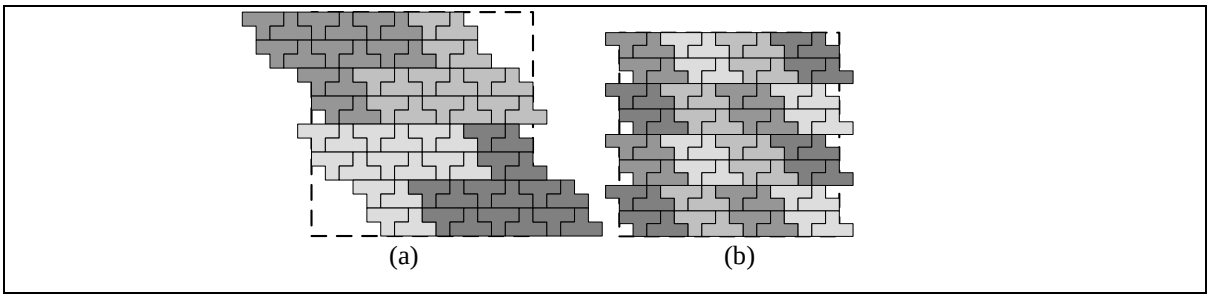


Şekil 18 Farklı boyutlar için döşeme gösterimleri.

Döşemelerin bir diğer özelliği ise donanım ile yazılım mimarilerinin birlikte tasarımına olanak kılmasıdır. Tasarımcının döşemeler arasında ya da döşeme içinde bulunan iletişimleri belirlemesi sayesinde, en kısa yoldan iletişimin belirlenmesi ve çakışmadan kaynaklı gecikmelerin önlenmesi sağlanabilir. Temel döşeme üzerinde yapılan bu tasarım, büyük boyutlara ölçeklendiğinde aynen korunur. Yazılım ile donanımın birlikte tasarlanması aynı zamanda enerji tüketimi gibi donanıma has özelliklerin eniyileştirilmesine de yardımcı olur. Örneğin bir örüntüde kullanılmayan iletişim yolları üzerindeki ağ birimlerinin kapatılması sağlanabilir.

Döşemeler bir kez oluşturulduktan sonra alan uzmanı tarafından yeni döşemelerin oluşturulması sağlanabilir. Şekil 19'da gösterildiği gibi yeni döşemeler, bir döşemenin döndürülmesi, özyineli ve/veya tekrarlayıcı biçimde ölçeklenmesi ile oluşturulabilir. Ayrıca tasarımcı farklı tipte döşemeleri birleştirerek yeni döşemeler oluşturabilir.

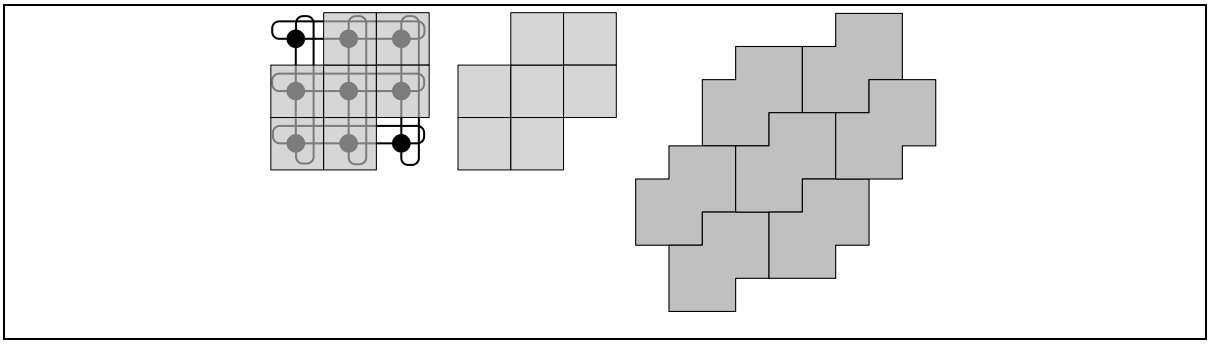
Döşemeleri daha büyük döşemelere ölçeklemek için iki değişik yöntem, özyineli döşeme ve tekrarlayıcı döşeme, kullanılmaktadır (Şekil 19). Bu döşeme yöntemleri 2B torus üzerinde yerleştirmeyi sağlamaktadır. Bu döşemeler ve iletişim yolları arasındaki ilişki de ileri kesimlerde anlatılacak olan iletişim örüntüleri ile tanımlanacaktır.



Şekil 19 16x16 torus için döşeme örnekleri: (a) Özyineli döşeme ve (b) tekrarlayıcı döşeme.

Bu iki yöntem kullanılarak yapılan ölçeklendirme, iletişimlerin çakışmadan bağımsız ve kilitlenmeden bağımsız olmasını sağlamaktadır. Küçük olan temel döşeme üzerinde bu durumları sağladığından emin olan alan uzmanı, ölçeklendirme sonucunda iletişim yollarının da aynı ölçüde ölçeklenmesi sayesinde, büyük sistemde de çakışmaların ve kilitlenmelerin olmadığından emin olur. Bu durum aynı zamanda döşeme kullanımının koşut sistemde yerellik ilkesini de yerine getirmesini sağlamaktadır.

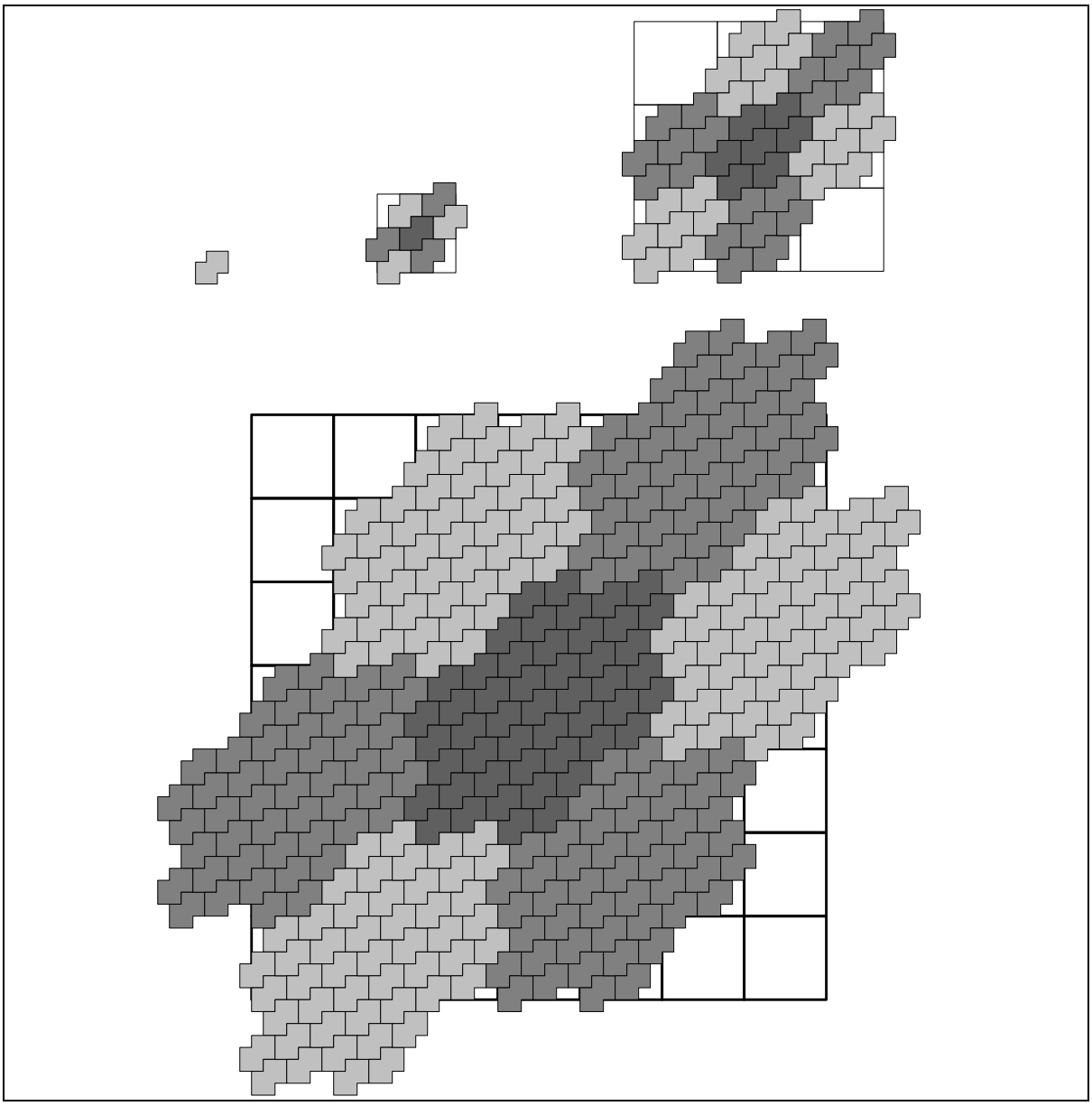
Döşemeler farklı topolojilerde farklı şekillerde oluşturulabilir. Oluşturulan döşemeler genellikle topolojiyi tam kaplayacak biçimde kullanılır, ancak her zaman kaplamak zorunda da değildir. Örneğin Şekil 20'de 7 düğümden oluşan bir döşeme gösterilmiştir. Bu döşeme 3x3 torus yapısını kapatamamaktadır, ancak bu döşeme özyineli olarak ölçeklendirildiğinde 7x7 torus yapısını tam olarak kapatabilmektedir (Şekil 20).



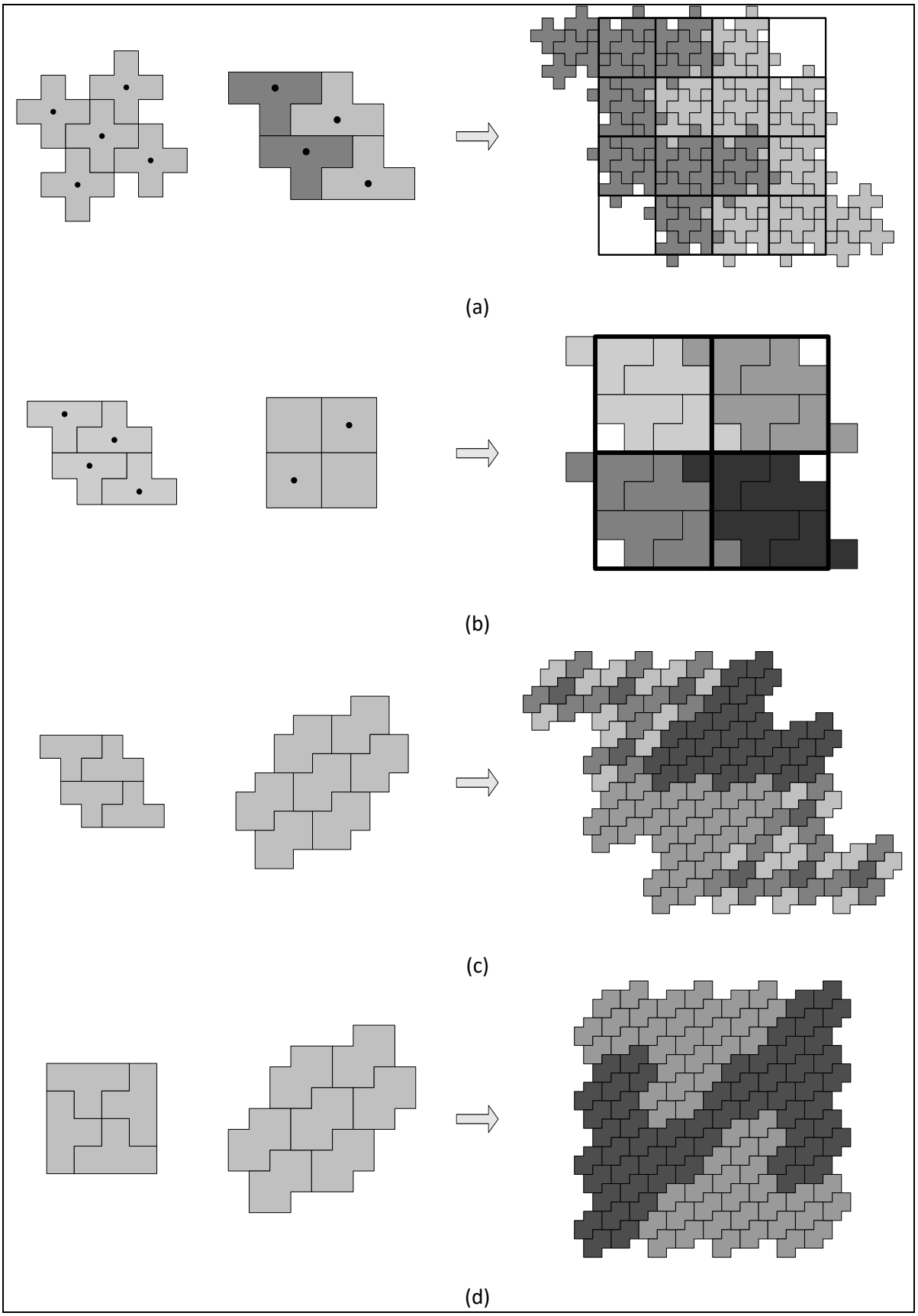
Şekil 20 7-işlemcili bir döşemeden özyineli olarak 49-işlemcili döşemenin oluşturulması.

Bu döşemeler yine özyineli olarak ölçeklendirilerek daha büyük boyutlarda oluşturulabilir. 7 düğümlü döşeme ile özyineleme ile 7x7 torus kaplanabilir. Bu döşeme ölçeklendirildiğinde 343 düğümlü döşeme oluşturulur ve yine özyineleme

kullanılarak 49x49 torus kaplanır (Şekil 21). Bu yöntem ile kaplanan torus üzerinde Strassen'in matris çarpma algoritmasının koşut gerçekleştirilmesi sağlanır [29]. Şekil 21 Strassen'in matris çarpmasında kullanılan 2B torus döşemelerinin parçalanması verilmiştir [30].



Şekil 21 Strassen'in koşut sisteminin özyineli parçalanması.

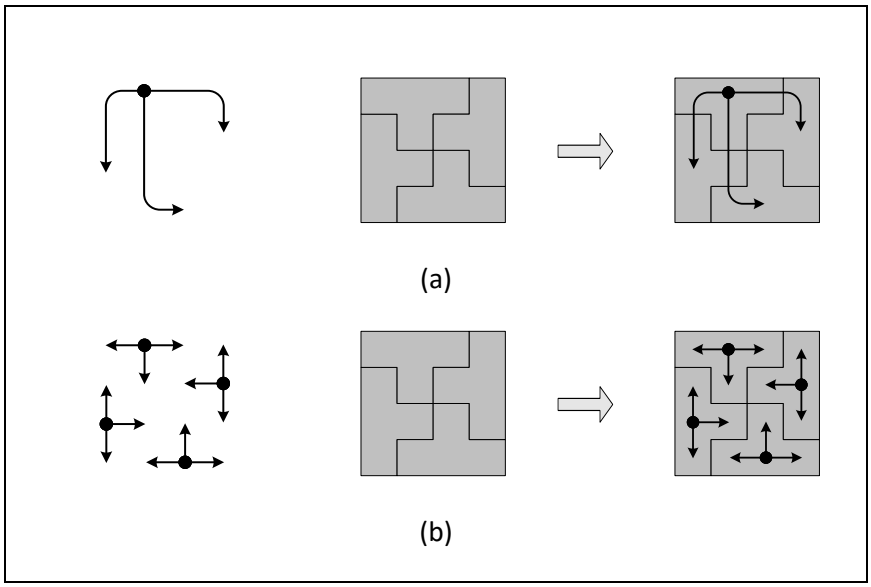


Şekil 22 Çeşitli boyutta yapıların oluşturulması için özyineli ve tekrarlayıcı döşemeler: (a) 5x5 ve 4x4 döşemelerden 20x20 döşemenin oluşturulması, (b) 4x4 ve 2x2 döşemelerden 8x8 döşemenin oluşturulması, (c) (d) 4x4 ve 7x7 döşemelerden 28x28 döşemenin oluşturulması.

Özyineli ve tekrarlı döşeme yöntemlerinin farklı boyutlarda döşemeler ile birlikte kullanımı, farklı boyutlarda döşemelerin oluşturulmasını sağlar. Şekil 22 (a)'da 5x5 ve 4x4 boyutlu döşemelerin özyineli kullanılması ile 20x20 boyutlu torus döşemesi oluşturulmuştur. Şekil 22 (b)'de ise 4x4 ve 2x2 boyutlu döşemeler ile 8x8 boyutlu torus döşemesi oluşturulmuştur. Şekil 22 (c) ve (d)'de ise 4x4 ve 7x7 boyutlu torus döşemesi ile özyineli ve tekrarlı yöntemi ile 28x28 torus boyutlu döşeme oluşturulmuştur.

4.1.2. İletişim Örüntüleri

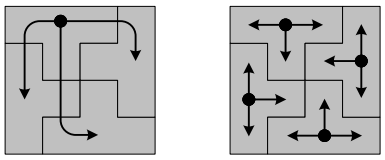
Bir iletişim örüntüsü, karşılık geldiği döşeme içinde atanmış olan iletişim yollarının kümesi olarak tanımlanabilir. Bir örüntü koşut verileri kullanacak olan düğümlerden oluşur ve bu düğümler birbiri arasında iletişim kurarlar. Bir iletişim hangi verinin hangi düğümden hangi düğüme gönderileceği bilgisini içerir. Bu kapsamda iletişim örüntüsü düğümler arasındaki ve düğüm içindeki iletişimi tanımlar (Şekil 23).



Şekil 23 Örüntüler ve uygun iletişim yollarından oluşturulan iletişim örüntüleri: (a) Hakim düğümler arasındaki iletişim örüntüsü ve (b) Hakim düğümlerden döşeme içindeki diğer düğümlere iletişim örüntüsü.

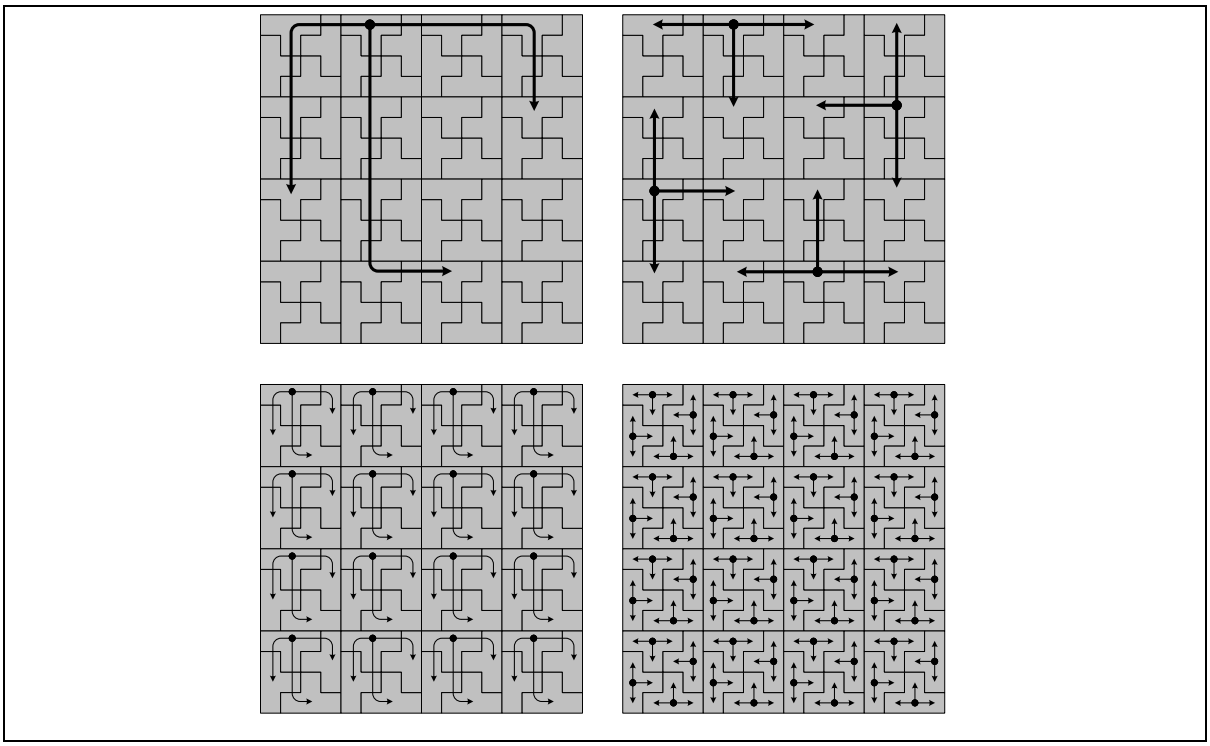
Koşut algoritmalar bu iletişim örüntüleri kullanılarak kurulabilir. Örneğin Şekil 24'de koşut bir sistem için birinden-hepsine (*one-to-all*) algoritmasının basitleştirilmiş hali gösterilmiştir. 4x4 bir torus üzerinde dört adet döşemeden oluşturulmuş bu koşut sistemde iki adet iletişim örüntüsü tanımlanmıştır. İlk adımda, bir düğüm veriyi diğer üç döşemenin hakim düğümüne gönderir. İkinci aşamada ise her hakim düğüm

kendi döşemesi içinde veriyi diğer düğümlerine dağıtır. Bu sayede ilk düğüm tüm veriyi sistemdeki diğer tüm düğümlere dağıtmış olur. Burada görüldüğü gibi her adım sıralı olarak gerçekleştirilmiştir. Her adım bir çalıştırma bölümü (*section*) olarak adlandırılabilir ve bu bölümler bu örnekte olduğu gibi sıralı olarak ya da koşut olarak çalıştırılabilir. Bir bölüm, hangi döşemenin ve iletişim örüntüsünün kullanılacağı ve doğal olarak nasıl bir sırada çalıştırılacağı bilgisine dayanarak, algoritmanın çalışma davranışını tanımlar. Eğer bölüm bir sırada çalıştırılacak ise sıralı, diğer bölümler ile eşzamanlı olarak hemen çalıştırılacak ise koşut olarak adlandırılır.



Şekil 24 4 adet T-şekilli döşeme ile oluşturulmuş 4x4 döşemesi için birinden-hepsine algoritması iletişim örüntüsü.

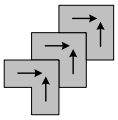
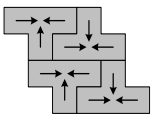
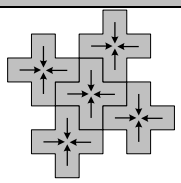
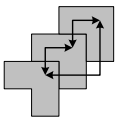
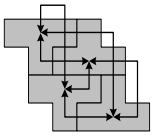
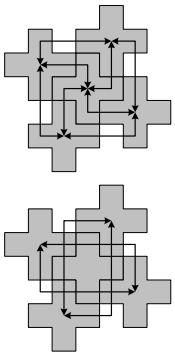
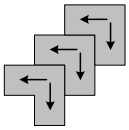
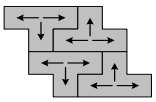
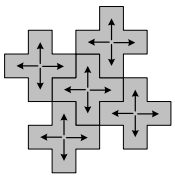
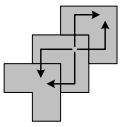
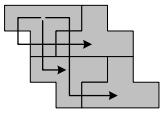
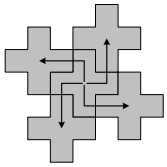
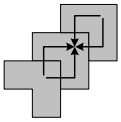
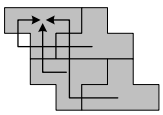
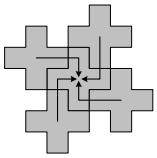
Verilen birinden-hepsine algoritması döşemesi ve iletişim örüntüsünün özyineli olarak ölçeklendirilmesi ile 16x16 torus üzerinde aynı algoritma kolaylıkla çalıştırılabilir. Büyük döşeme göz önüne alındığında T şeklindeki büyük döşemelerde ana düğüm diğer düğümlerin hakim düğümüne veriyi ilk iletişim örüntüsünde olduğu gibi gönderir. Daha sonra bu büyük döşemeler içinde ikinci iletişim örüntüsü çalıştırılarak daha küçük T şeklindeki düğümlere veriler gönderilir. Özyineli olarak bir alt seviyede küçük döşemelerin herbirinde yine önce ilk, sonra da ikinci iletişim örüntüsü çalıştırılır ve bu sayede birinden-hepsine algoritması tamamlanmış olur. Şekil 25, bu özyineli olarak çalışan birinden-hepsine algoritmasını göstermektedir. Ölçekleme daha büyük boyutlara taşındığında 64x64, 256x256... şeklinde daha büyük torus topolojilerinde çalıştırılabilir.



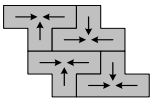
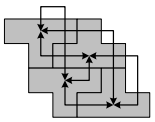
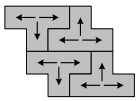
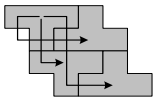
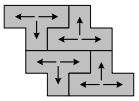
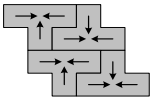
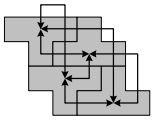
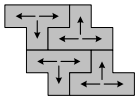
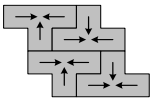
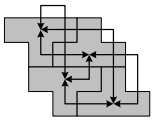
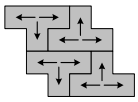
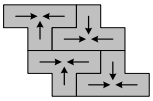
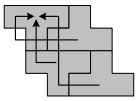
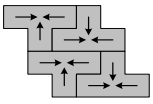
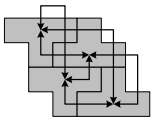
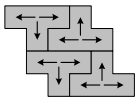
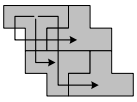
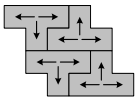
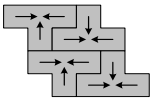
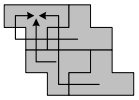
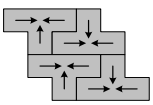
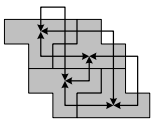
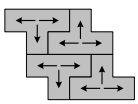
Şekil 25 Özyineli ölçeklendirilmiş 16x16 döşeme için birinden-hepsine iletişim örüntüsü.

Bu görsel gösterim ile istenilen şekilde tasarlanan algoritmaya uygun olarak döşemeler ve iletişim örüntüleri oluşturulabilir. Ancak farklı torus boyutları ve temel toplu işlem algoritmaları için döşemeler ve iletişim örüntüleri sınıflandırılabilir. Çizelge 1 dört farklı torus boyutu için temel iletişim örüntülerini göstermektedir. Temel olarak mesaj iletimleri hakim düğümler aracılığıyla yapılmaktadır. Örneğin parçalayıp dağıtma (*scatter*) algoritmasında veri hakim düğüm tarafından eşit parçalara ayrılarak döşeme içindeki diğer düğümlere gönderilir. Dağıtım (*multicast*) algoritması da benzer biçimde çalışır ancak veri parçalara ayrılmaz, toplu olarak gönderilir.

Çizelge 1 Hakim düğümler ile yönetilen düğümler arasındaki temel toplu işlemler

Temel Toplu İşlem İşlevleri	2×2	3×3	4×4	5×5
Gather				
Exchange				
Scatter				
Multicast				
Gather–to–One				

Çizelge 2 Algoritmalar

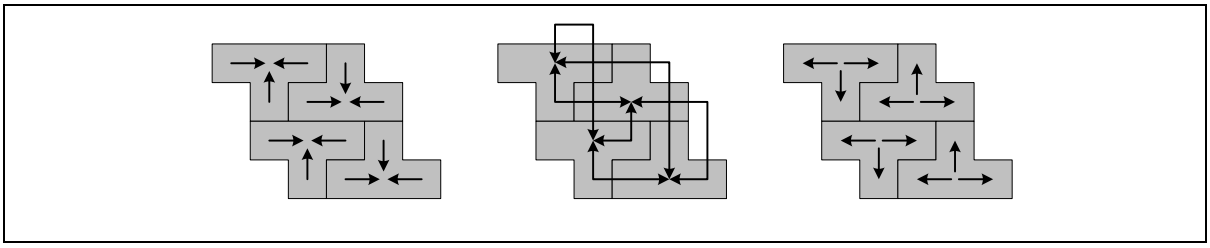
Koşut Algoritmalar	I	II	III
Barrier			
One-to-All Broadcast			
All-to-All Broadcast			
Complete Exchange			
Reduce			
All-Reduce			
Scatter			
Gather			
All-Gather			

Koşut algoritmalar ise bu tip iletişim örüntülerinin koşut olarak ya da sıralı olarak çalıştırılması ile gerçekleştirilir. Her adımda yapılacak iletişim örüntüsü tanımlandığında bu sıralı işlemeler işletilir. Çizelge 2'de örnek olarak verilen bir 4x4 torus döşemesi için farklı koşut algoritmalarda işletilen sıralı iletişim örüntüleri verilmiştir.

Çizelge 1 ve Çizelge 2'de verilen döşeme ve iletişim örüntüleri ile sıralı olarak işletilme adımları aynı döşemenin özyineli olarak ölçeklenmiş daha büyük boyutlu torus yapılarında da aynen kullanılabilir. Özyineli olarak büyütme sonucu bu algoritmalarda verilen iletişim örüntüleri orantısal olarak büyütüldüğü için çakışmadan bağımsız olarak tüm algoritmalar ölçeklenir.

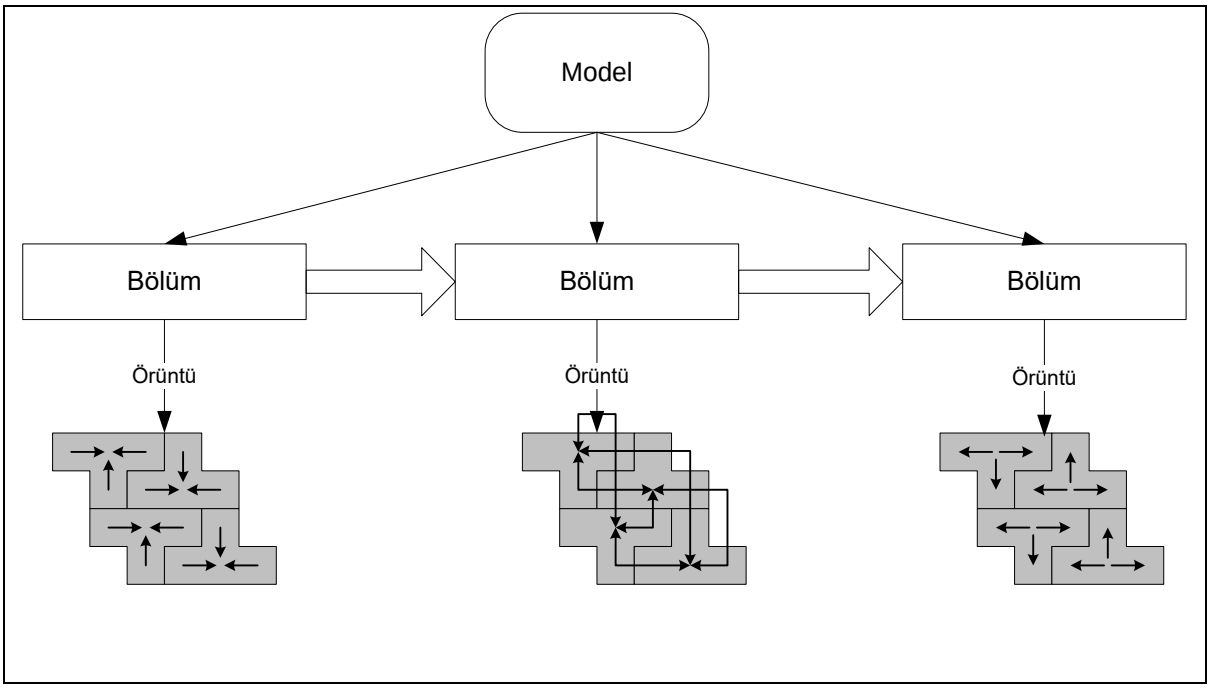
4.1.3. Çalıştırma Bölümleri

Döşemeler ve iletişim örüntülerinin birlikte koşut algoritmaların gerçekleştirilmesinde sonuç olarak bu iletişim örüntülerinin ilgili döşemeler üzerinde farklı çalışma bölümleri olarak çalıştırılması çıkarılabilir. Döşemeleri ve iletişim örüntülerinin hakim düğümler (*dominating node*) ve ona bağlı yönetilen düğümler (*dominated node*) olarak düşünürsek, iletişim örüntüleri noktadan-noktaya hakim düğümler arası ya da hakim düğümden yönetilen düğüme ileti geçişi (*message-passing*) olarak düşünebiliriz. Her iletişim örüntüsü de bir çalıştırma bölümü olarak değerlendirilebilir. Şekil 26'da 4x4 torus için tüm-değişim (*complete-exchange*) algoritmasının adımları verilmiştir.



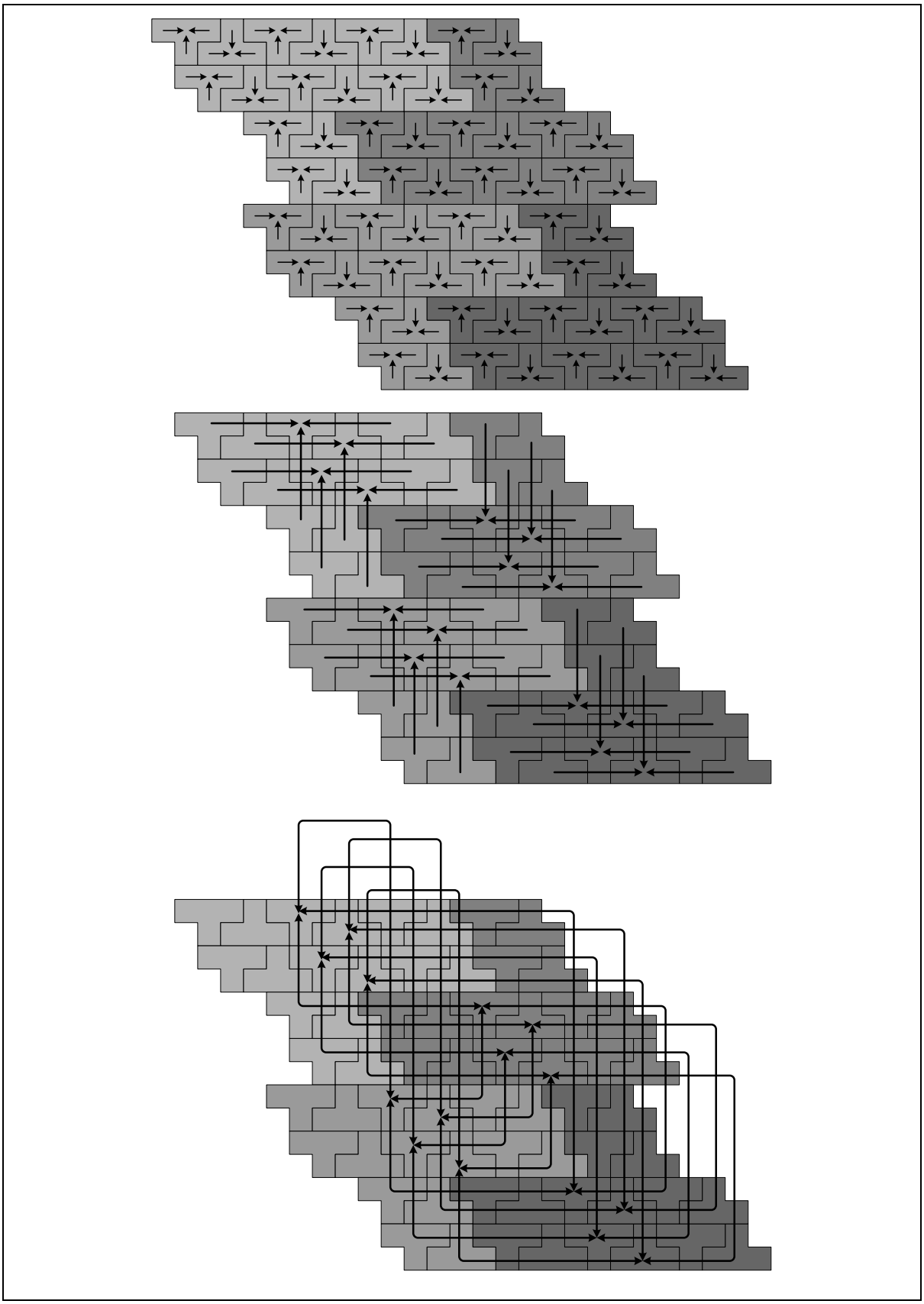
Şekil 26 4x4 döşeme kullanarak tüm-değişim algoritması.

Burada koşut algoritmanın ileti geçişi dışında yapılacak seri kod kesimlerini de karşılaması gerektiği ortaya çıkmaktadır. Bu seri kod kesimleri faaliyet (*action*) olarak adlandırabiliriz. Bu tip faaliyetler koşut algoritmalarda bir ileti geçişinin öncesinde, ya da sonrasında ya da o ileti geçişi yapıldığı sırada olabilir. Dolayısıyla oluşturulan iletişim örüntüleri bu faaliyetleri ve ne zaman çalıştırılacağı bilgisini içerecektir. Bu iletişim örüntüleri ise daha önce belirtildiği gibi sıralı ya da koşut çalıştırma bölümlerinde tanımlanır. Örnek olarak verilen tam-değişim algoritmasının bu çalışma kesimleri ile ifade edilmesi gerektiğinde Şekil 27'deki gibi bir gösterim oluşturulacaktır.

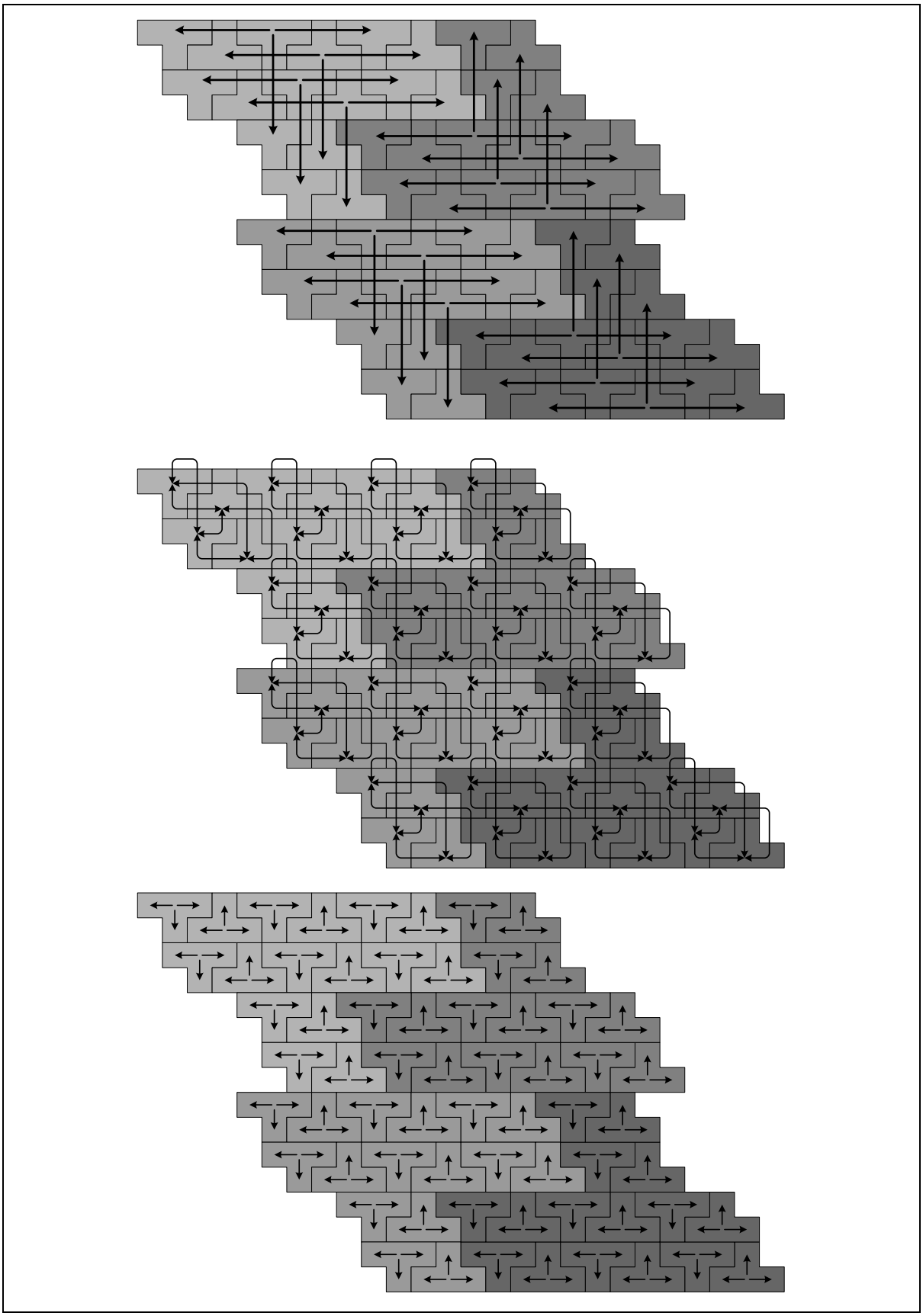


Şekil 27 4x4 torus üzerinde tüm-değişim örneğinin çalıştırılması için sıralı bölümler.

Tam değişim algoritması daha büyük boyutlara özyineleme yöntemi kullanılarak ölçeklenebilir. Şekil 26'da verilen gösterimdeki adımlar özyineli olarak çoğaltıldığında16x16 torus için döşemeler ve iletişim örüntüleri Şekil 28 ve Şekil 29'da gösterildiği gibi elde edilmektedir.



Şekil 28 16x16 torus için tüm-değişim algoritması (ilk üç adım).



Şekil 29 16x16 torus için tüm-değişim algoritması (son üç adım).

4.2. MODEL GÜDÜMLÜ YAKLAŞIM

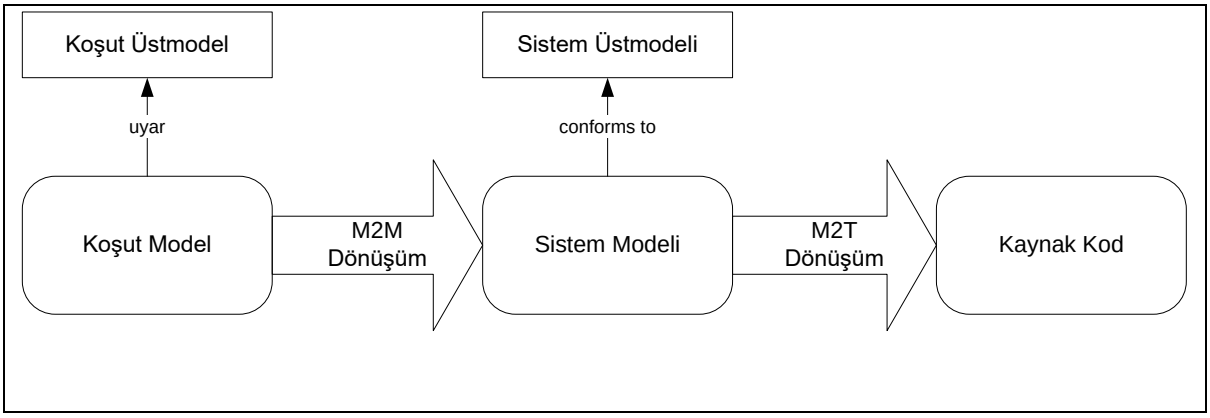
Bilgisayar bilimlerinde dil mühendisliği kavramı düşünüldüğünde koşut sistem için alana özgü dilin koşut sistemin tasarımı, gerçekleştirmesi, doğrulaması ve geçerlemesi gibi konularda destek vermesi gerekmektedir [31]. Tasarım, gerçekleştirme, doğrulama ve geçerleme için araç geliştirmede görsel dillerin kullanımı önemlidir. Özellikle gerçekleştirme düzeyinde ana kavramların oluşturulması ve kod oluşturma işlemlerinin sağlanmasında alana özgü görsel dil koşut alanındaki bilgiyi ve deneyimi kullanır [32].

Model Güdümlü Yazılım Geliştirme (MGYG) yaklaşımı koşut yazılım geliştiricilerine sistemi kolaylıkla tanımlamalarında yardımcı olur. Görsel dil desteği sağlayan bir MGYG aracı bu aşamadaki ileri ve tersine mühendislik (*forward and reverse engineering*) gereksinimlerini karşılayacaktır. Bu yaklaşımı kullanmak amacı ile sistemin gereklerini belirlemeyi sağlayan soyut ifadesi [33] olan koşut sistem modeli tanımlanmalıdır. Bu model program kodunun algoritma ve bileşenleri arası iletişimlerin belirlenmesi gibi soyutlamasına kaynak olacaktır. Ancak model sistemin kendisi değil, soyut bir gösterimi olduğu unutulmamalıdır.

MGYG yaşam döngüsü beş adımdan oluşmaktadır [34]. Öncelikle tüm gereksinimlerin ele alınabilmesi ve bu gereksinimleri kapsayacak bir alana özgü dilbilgisinin oluşturulması gerekmektedir. Bu dilbilgisi aynı zamanda modelimizin dayandırıldığı bir üst model (*metamodel*) olarak tanımlanabilir. Üst model bu alana özgü görsel dilimizin görsel gösterimlerini tanımlayacaktır. Oluşturulan olgular, ya da diğer bir adıyla modeller, bu gösterime uygun biçimde sistem tanımını yapar [35]. Bir sonraki adım ise platformdan bağımsız modelin (*platform independent model* - PIM) oluşturulmasıdır. Bu model hedeflediğimiz platformun özelliklerinden bağımsız olarak soyut tanımları içerecektir. Platformdan bağımsız modelden sonra, bu modelden platforma özel model oluşturulacaktır. Platformdan bağımsız modelden platforma özgü modelin oluşturulması için modelden-modele dönüşüm kullanılacaktır. Sonraki aşamada, modelden-metne dönüşüm kullanılarak platforma özgü modelden kaynak kodun üretilmesi sağlanır. Son aşama ise oluşturulan kaynak kodun hedef platform üzerinde çalıştırılmasıdır (*deployment*).

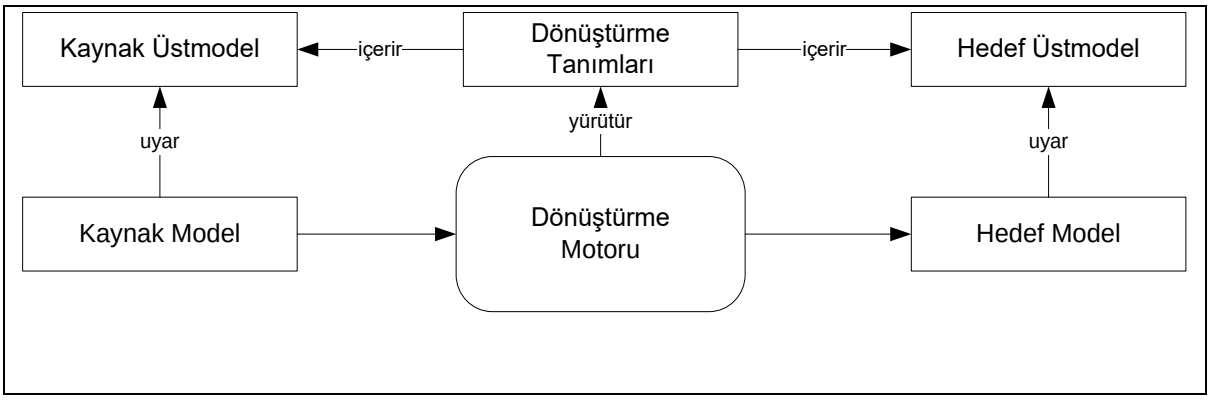
Burada model güdümlü yaklaşımımızda, koşut sisteme ait platformdan bağımsız modelimiz gösterimde belirtilen döşeme ve iletişim örüntüsü bilgilerini içerecektir.

Hedef sistemde çalıştırılacak olan kaynak kodun üretilmesi amacıyla Şekil 30'da verilen dönüşüm zinciri uygulanabilir. Burada sistem modeli ve üst modeli koşut sistem üzerinde kullanılacak olan sistem kütüphanesine göre belirlenecektir. Bu sistem modeli MPI gibi bir kütüphaneyi ya da gelecekte oluşturulacak aşırı ölçekli sistemler için bir kütüphane ya da dili içerebilir. Kaynak kodun üretilmesi de bu sistem modelinin kullanılması ile gerçekleştirilecektir.



Şekil 30 Koşut sistemler için model-güdümlü dönüşüm zinciri.

Dönüşüm işlemi hem kaynak üst modeli hem de hedef üst modeli karşılayan dönüşüm kuralları ile yapılmaktadır. Model dönüşümünün temel davranışı Şekil 31'da verilmiştir. Dönüşüm kuralları kaynak modeldeki bileşenlerin hedef modele ne şekilde yansıyacağını belirleyen kurallardır. ATL, ETL, Operational QVT gibi dönüşüm motorları bu amaçla kullanılabilir.



Şekil 31 Model dönüşümü kavramsal gösterimi.

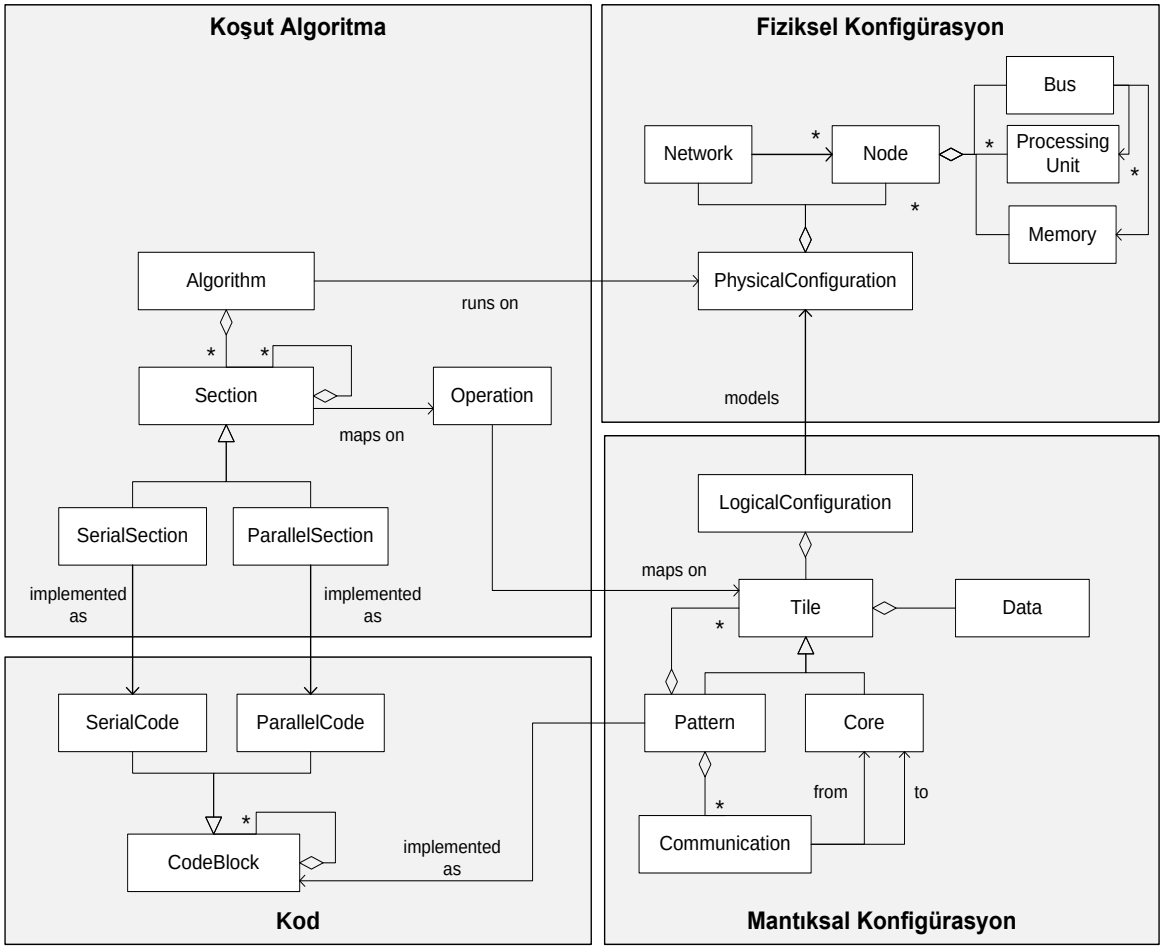
Hedef model oluşturulduktan sonra kaynak kodun oluşturulması için modelden-metne dönüşüm kuralları uygulanacaktır. Modelden-metne dönüşüm kuralları

genelde şablon kullanılarak gerçekleştirilmektedir. Xpand, EGL gibi modelden-metne dönüşüm motorları kullanılabilmektedir.

4.2.1. Koşut Üst Model

Alana özgü dil, model güdümlü geliştirmenin temel taşıdır [36]. Alana özgü dil, geliştiricilerin problemin çözümündeki anahtar noktaların belirlenmesine yardımcı olur. Görsel bir alana özgü dilin alan uzmanlarına verdiği en önemli fayda, genel olarak kabul görmüş kavram, gösterim ve kurallar kümesi kullanarak problemin ve çözümünün daha iyi ifade edilebilmesi amacıyla sistemin basitleştirilebilmesidir [37]. Alan uzmanı sistemin davranışını modelleyebilir ve bu alana özgü dil ile oluşturulan modelin geçerlenmesini sağlayabilir. Buradan model güdümlü geliştirmede kullanılan model dönüşümleri ve kod üretilmesi işlemlerine imkan sağlar. Bu sebeple alana özgü dilin oluşturulması koşut sistemlerin model güdümlü geliştirilmesi için önemlidir.

Koşut algoritmaların koşut hesaplama platformlarına atanması için model güdümlü yazılım geliştirme yaklaşımının oluşturulabilmesi için koşut üst modelinin tanımlanması gerekmektedir. Şekil 32 koşut işlem için tanımlanmış olan üst modeli göstermektedir. Üst modelde koşut algoritmaların koşut hesaplama platformlarına atanması için gereken dört ayrı konsepti içermektedir. Bu konseptler koşut algoritma, fiziksel konfigürasyon, mantıksal konfigürasyon ve kod olarak sıralanmaktadır.



Şekil 32 Koşut işlem üst modeli.

Koşut Algoritma kesiminde algoritmanın (*Algorithm*) birden çok çalıştırma bölümünden (*Section*) oluştuğu görülmektedir. Çalıştırma bölümleri sıralı (*SerialSection*) ya da koşut (*ParallelSection*) olabilir. Her çalıştırma bölümü mantıksal kesimde bir döşemeye (*Tile*) adreslenmiş bir işleve (*Operation*) atanmıştır.

Fiziksel konfigürasyon kesimi koşut hesaplama platformunun fiziksel konfigürasyon bilgisini içermektedir. Bir fiziksel konfigürasyon bir ağ (*Network*) ve düğümlerden (*Node*) oluşmaktadır. Ağ, düğümler arasındaki iletişim ortamını tanımlamaktadır. Bir düğüm veri yolu (*Bus*), bellek (*Memory*) ve işlem birimlerinden (*ProcessingUnit*) oluşur. Birden fazla sayıda işlem birimi ve bellek kullanımı ile paylaşımlı bellekli ya da dağıtık bellekli mimarilerde çok çeşitli fiziksel konfigürasyonların tanımlanması mümkün olmaktadır.

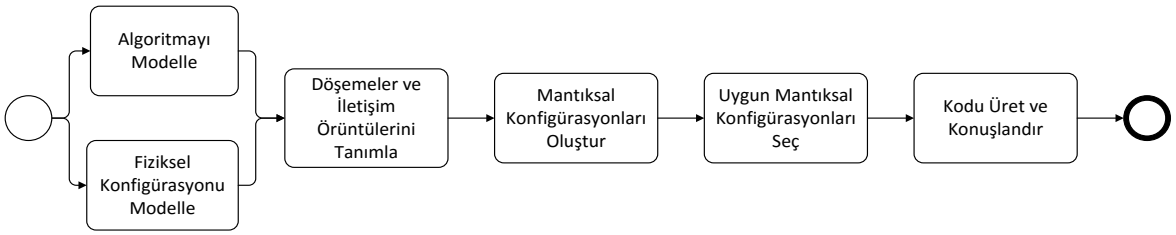
Mantıksal konfigürasyon fiziksel konfigürasyona bağlı olarak düğümler arasındaki mantıksal iletişim yapılarının bir modelini göstermektedir. Mantıksal konfigürasyon, döşemelerden (*Tile*) oluşmaktadır. Bir döşeme, tek bir çekirdek (*Core*) ya da döşemelerden oluşan bir örüntüden (*Pattern*) oluşmaktadır. Çekirdekler arasındaki iletişim bağlantıları örüntüyü oluşturur ve bir örüntü bir kod kesimi (*CodeBlock*) olarak gerçekleştirilir. Algoritma bölümleri bu kod kesimlerine adreslenmektedir. Burada, bir sıralı bölüm bir sıralı kod (SerialCode) ile, koşut bölüm ise bir koşut kod (*ParallelCode*) ile gerçekleştirilir. Son olarak da bir algoritma (*Algorithm*), bir fiziksel konfigürasyon (*PhysicalConfiguration*) üzerinde çalıştırılacaktır.

4.2.2. Model Güdümlü Yazılım Geliştirme Yaklaşımı

Koşut üst modelin tanımlanması sonrası, bu üst model kullanılarak koşut algoritmaların koşut hesaplama platformlarına atanması için kullandığımız model güdümlü yazılım geliştirme yaklaşımının tanımlanması gerekmektedir. Şekil 33 BPMN [38] sözdizimi kullanılarak tanımlanan yaklaşımı sergilemektedir. Yaklaşımda kullanılmak üzere altı temel süreç tanımlanmıştır. Bu süreçler:

- Algoritmayı Modelle,
- Fiziksel Konfigürasyonu Modelle,
- Döşemeler ve İletişim Örüntülerini Tanımla,
- Mantıksal Konfigürasyonları Oluştur,
- Uygun Mantıksal Konfigürasyonları Seç,
- Kodu Üret ve Konuşlandır,

olarak tanımlanmaktadır. Her bir süreç için tanımlanan üs modele uygun olarak farklı modeller tanımlanmaktadır. Bu modellerin tanımlanmasında ISO/IEC 42010 [39] uygun olarak tasarım görünümleri uygulanmıştır [40]. Sistemin modellenmesinde farklı bakış açıları tanımlanarak farklı görünümlerde modelleme yapılmasına imkan sağlanmıştır.



Şekil 33 Koşut algoritmaların koşut hesaplama platformlarına atanması için model güdümlü yazılım geliştirme yaklaşımı.

Yaklaşımın gerçekleştirilmesinde bir yazılım geliştirme ortamında bu biçimsel görünümlerin tanımlanması sağlanmıştır. Ecore [24], Üst Nesne Olanağı (*Meta-Object Facility* - MOF)[25] için geliştirilmiş ve Eclipse [41] yazılım geliştirme ortamı ile kullanılan üst model tasarlamak için uygun bir uygulamadır. Eclipse Modelleme Çatısı (*Eclipse Modelling Framework*) [24] alana özgü dil olarak kullanılan üst modellerin oluşturulması için geliştirme araçları sağlamaktadır.

Modelleme ortamını tanımlamak için önceki kesimlerde anlatılan bileşenlerin tanımlanması gerekmektedir. Bu ortam koşut alana budayacak araç desteği sunarak koşut problemlerin çözümüne yardımcı olmayı amaçlamalıdır [37].

Bu kesimde yaklaşımda tanımlanan altı sürecin detayları anlatılmaktadır.

4.2.2.1. Algoritmayı Modelle

Koşut algoritmanın koşut hesaplama platformuna atanması için öncelikle algoritmanın çözümlenmesi ve parçalanan algoritma bölümlerinin sıralı mı koşut mu olduğunun kararının verilmesi gerekmektedir. Algoritmanın her bölümü komut kümesinin soyut ifadesi olan bir işlevi gerçekleştirir. Sıralı bölüm, sıralı olarak çalıştırılması gereken kod kesimlerini içerir. Koşut bölüm ise ilkel koşut işlevleri gerçekleştirir. Örneğin, bir koşut bölüm ilkel işlevlerden *Scatter* işlevini diğer düğümlere dağıtım amaçlı gerçekleştirirken diğer bir bölüm diğer düğümlerden verileri toplamak için *Gather* işlevini gerçekleştirir, veriyi tüm düğümlere dağıtmak için de *Broadcast* işlevini gerçekleştirir vs. Bu çözümleme sonuçlarını ifade edebilmek ve algoritmayı modelleyebilmek için koşut üst modelimizin Koşut Algoritma kesimine uygun olarak Çizelge 3'te verilen algoritma parçalama bakış açısı tanımlanmıştır.

Çizelge 3 Algoritma Parçalama Bakış Açısı

<table>
<tr><td>Adı</td><td>Algoritma Parçalama Bakış Açısı</td></tr>
<tr><td>İlgisi</td><td>Algoritmanın sıralı ya da koşut olarak farklı bölümlere parçalanması. Algoritmanın çözümlenmesi.</td></tr>
<tr><td>Paydaşları</td><td>Algoritma çözümleyici, mantıksal konfigürasyon mimarı, fiziksel konfigürasyon mimarı</td></tr>
<tr><td>Öğeleri</td><td>Algorithm – bölümlerden oluşan koşut algoritma
Serial Section (SER) – sıralı olarak çalışması planlanan komut kümesinden oluşan algoritma parçacığı
Parallel Section (PAR) – koşut olarak çalışması planlanan komut kümesinden oluşan algoritma parçacığı
Operation – algoritma bölümünün çalıştırdığı komut kümesinin soyut ifadesi</td></tr>
<tr><td>İlişkileri</td><td>Parçalama ilişkisi – algoritma ve bölümlerini tanımlar</td></tr>
<tr><td>Kısıtları</td><td>Bir bölüm ya SER ya da PAR olabilir, ikisi birden olamaz.</td></tr>
<tr><td>Gösterimi</td><td><table><tr><td>Index</td><td>Algorithm Section</td><td>Section Type</td><td>Operation</td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr></table></td></tr>
</table>

Tanımlanan bakış açısı kullanarak verilen bir algoritmanın algoritma parçalama görünümü oluşturulabilir. Çizelge 4'de örnek bir matris çarpma algoritması için algoritma parçalama görünümü verilmektedir. Bu algoritma dört farklı bölüme parçalanmıştır. İlk bölüm alt matrislerin farklı düğümlere dağıtılmasını sağlayan bölümdür. Bu bölüm koşut (PAR) olarak işaretlenmiştir. İkinci bölüm ise sıralı (SER) olarak işaretlenmiş ve alt matrislerin çarpımını ifade etmektedir. Çarpım sonuçları üçüncü bölümde tekrar koşut olarak toplanmaktadır. Bu sebeple bu bölüm koşut (PAR) olarak işaretlenmiştir. Son bölümde ise toplanan matris sonuçları sıralı (SER) olarak toplanmakta ve sonuç üretilmektedir.

Çizelge 4 Algoritma Parçalama Görünümü

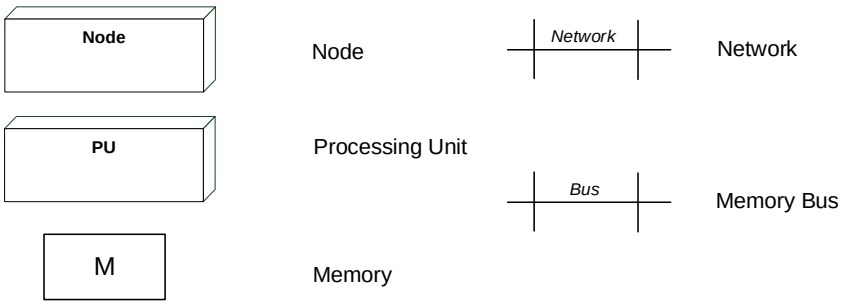
İn.	**Algorithm Section**	**Section Type**	**Operation**
1	Alt matrisleri dağıt	PAR	Scatter
2	C = A * B	SER	-
3	Matris çarpım sonuçlarını topla	PAR	Gather
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	SER	-

4.2.2.2. Fiziksel Konfigürasyonu Modelle

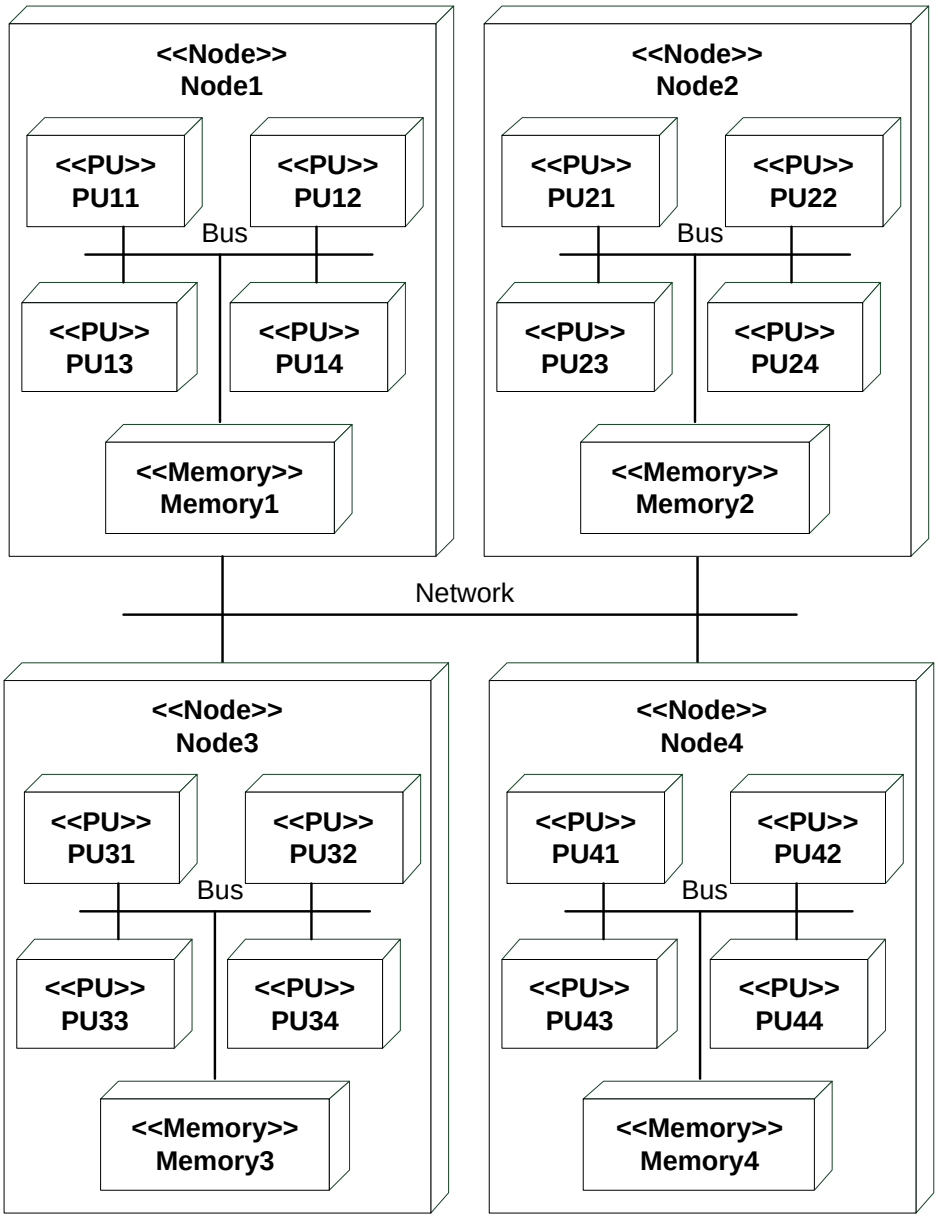
Koşut algoritmanın çözümlenmesi ve parçalanması sonrasında koşut hesaplama platformu için fiziksel konfigürasyonun modellenmesi gerekmektedir. Çizelge 5 bu

amaçla oluşturulan fiziksel konfigürasyon bakış açısını göstermektedir. Bu bakış açısı koşut üst modelin fiziksel konfigürasyon kesimine uygun olarak tanımlanmıştır.Düğümler, ağ, bellek, işelm birimleri gibi gösterimleri içermektedir.

Çizelge 5 Fiziksel Konfigürasyon Bakış Açısı

Adı	Fiziksel Konfigürasyon bakış Açısı
İlgisi	Koşut hesaplama platformunun fiziksel konfigürasyonunun tanımlanması
Paydaşları	Fiziksel Konfigürasyon Mimarı
Öğeleri	Node – Birden fazla işlemci, bellek ve veri yolu içerebilen basit bir bilgisayar mimarisi Network – Düğümleri bağlayan iletişim ortamı Memory Bus – Bir düğümdeki işlem birimleri ve bellekleri bağlayan iletişim ortamı Processing Unit – Komut kümelerini çalıştıran işlem birimi Memory – veri saklamak için bellek
İlişkileri	Düğümler ağ ile birbirleri ile iletişim kurarlar İşlem birimleri ve bellekler veri yolu ile birbirlerine bağlanır
Kısıtları	İşlem birimleri sadece düğümler içinde olabilir Bellek paylaşımlı ya da dağıtık yapıda olabilir
Gösterimi	

Şekil 34 örnek bir fiziksel konfigürasyon görünümü vermektedir. Burada fiziksel konfigürasyon dört adet düğümden ve bu düğümler arasında oluşturulmuş bir adet ağdan oluşmaktadır. Her düğümün dört adet işlem birimi düğüm içinde paylaşımlı bir bellekten oluşmaktadır. Bu birimler bir veri yolu ile bağlanmıştır.

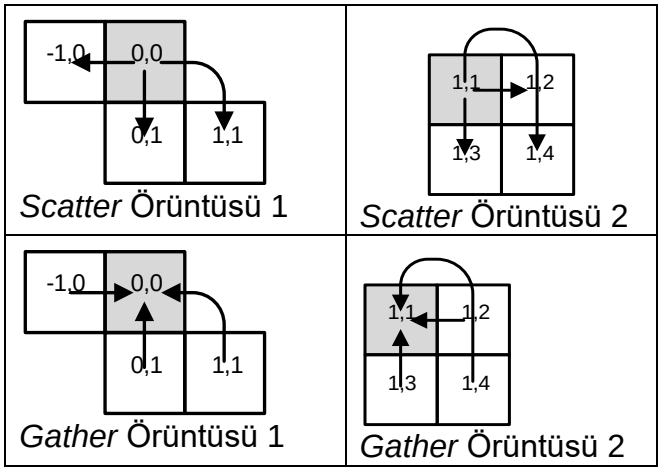


Şekil 34 Fiziksel Konfigürasyon Görünümü

4.2.2.3. Döşemeler ve İletişim Örüntülerini Tanımla

Daha önce belirtildiği gibi koşut işlevler (Çizelge 1) birden fazla işlem birimi üzerinde çalıştırılırlar. Bunlara örnek *Scatter* işlevinin verinin birden fazla birime dağıtması ya da *Gather* işlevinin birden fazla birimden verinin toplanması verilebilir. Bu tip koşut işlevler bilinen döşeme ve iletişim örüntüleri ile tanımlanabilir. Şekil 35'de matris çarpma algoritması için örnek *Gather* ve *Scatter* döşeme ve iletişim örüntüleri verilmiştir. Matris çarpma algoritmasında çarpılması gereken 8 adet alt matris dört adet düğüme ikişer ikişer *Scatter* işlevi ile dağıtılmakta, çarpım sonuçları da *Gather*

işlevi ile toplanıp sıralı olarak toplama işlemine sokulmaktadır. Burada iki işlev için de iki farklı döşeme ve iletişim örüntüsü tanımlanmıştır.

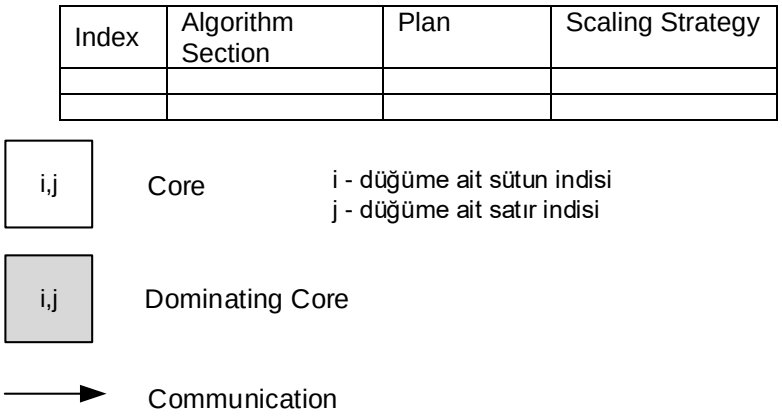


Şekil 35 Örnek *Gather* ve *Scatter* işlevleri döşeme ve iletişim örüntüleri.

Döşeme tanımlamalarının yapılması için çarpanlarına ayırma [42] yöntemi kullanılabilir. Çarpanlarına ayrılmış değerler fiziksel konfigürasyonun boyutu ile doğru orantıda ölçekleme çarpanı olarak kullanılır. Ölçekleme çarpanı bir fiziksel konfigürasyonun boyutunun diğer bir fiziksel konfigürasyonun boyutuna oranı olarak tanımlanabilir. Örneğin ölçeği 6x6 olan bir fiziksel konfigürasyonun 2x2 boyutundaki fiziksel konfigürasyona ölçekleme çarpanı 3'dür. Bu aynı zamanda 3x3 boyutlu bir döşeme ile 2x2 boyutlu bir döşemeyi birlikte ölçekleyerek 6x6 boyutunda bir fiziksel konfigürasyon oluşturabiliriz. Özyineli biçimde 2x2 boyutlu döşeme ile de 12x12 boyutlu fiziksel konfigürasyonu oluşturabiliriz.

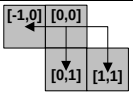
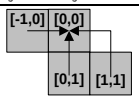
Bu modellemeyi desteklemek için yaklaşımımızda algoritmadan-mantıksal konfigürasyona atama bakış açısı tanımlanmıstır (Çizelge 6). Bakış açısı her algoritma bölümü için kullanılacak olan döşeme ve iletişim örüntüsünü göstermektedir.

Çizelge 6 Algoritmadan-Mantıksal Konfigürasyona Atama Bakış Açısı

Adı	Algoritmadan-Mantıksal Konfigürasyona Atama Bakış Açısı
İlgisi	Algoritmanın mantılsal konfigürasyona atanması için iletişim örüntülerinin tanımlanması
Paydaşları	Sistem Mühendisleri, Mantıksal Konfigürasyon Mimarı
Öğeleri	Section – Bir algoritmanın belli bir komut kümesini içeren bölümü. Bu bölüm sıralı ya da koşut olabilir. Plan – Her bölüm için işlevlerin mantıksal konfigürasyona atanması için gereken plan Core – işlem birimi modeli Dominating Core – Diğer düğümler ile veri alışverişinden sorumlu hakim işlem birimi Communication – işlem birimleri arası veri haberleşmesi
İlişkileri	Plana göre işlevlerin mantıksal konfigürasyona atanması
Kısıtları	Her sıralı bölümün hangi düğümlerde çalışması gerektiği planda belirtilir. Her koşut bölüm ilgili döşeme ve iletişim örüntüsüne göre bir planı tanımlanır.
Gösterimi	

Tanımlanan bu bakış açısına göre oluşturulan örnek bir algortimadan-mantıksal konfigürasyona atama görünümü Çizelge 7'de verilmiştir. Tanımlanmış bölümlere ek olarak ilerleyen süreçte mantıksal konfigürasyonun oluşturulması için plan ve ölçekleme stratejisi bu görünümde verilmektedir. Plan koşut bölümler için ilgili işleve uygun olarak tanımlanan döşeme ve iletişim örüntüsünü göstermektedir. Sıralı bölümler için ise nasıl çalıştırılacağı hakkında bilgiyi vermektedir. Örnekte sıralı bölümlerin her düğümde çalıştırılacağı belirtilmektedir. Koşut bölümler için ise plan ve ölçekleme stratejileri belirtilmiştir. Ölçekleme stratejisi işlevin özelliğine göre aşağıdan-yukarıya (UP) ya da yukarıdan-aşağıya (DOWN) olarak belirlenebilir. Bu seçilen ölçekleme çarpanlarının hangi sırada kullanılacağını göstermektedir.

Çizelge 7 Algoritmadan-Mantıksal Konfigürasyona Atama Görünümü

In.	Algorithm Section	Section Type	Plan	Scaling Strategy
1	Alt matrisleri dağıt	PAR		DOWN
2	C = A * B	SER	Her düğümde çalışacak	N/A
3	Matris çarpım sonuçlarını topla	PAR		UP
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	SER	Her düğümde çalışacak	N/A

4.2.2.4. Mantıksal Konfigürasyonları Oluştur

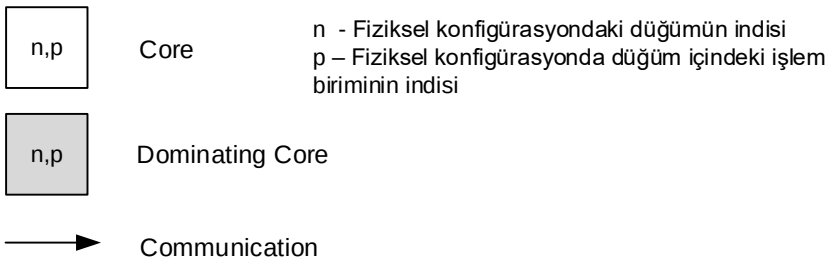
Algoritmanın mantılsal konfigürasyona nasıl atanacağı ile ilgili döşemelerin ve iletişim örüntülerinin tanımlanması ve planın oluşturulmasından sonra mantıksal konfigürasyonların oluşturulması sağlanabilir. Oluşturulacak mantıksal konfigürasyon koşut üst modelin mantıksal konfigürasyon kesimine uygun olarak oluşturulmaktadır. Mantıksal konfigürasyon modelinin oluşturulması için Şekil 36'de verilen algoritma tasarlanmıştır. Mantıksal konfigürasyon modeli oluşturulurken algoritma bölümleri kullanılmaktadır. Algoritma bölümü sıralı (SER) olduğunda ilgili komut kümesi modele aynen yansıtılmıştır. Algoritma bölümü koşut (PAR) olduğu durumda kullanılacak olan plandaki iletişim örüntüsü bulunur, bu örüntünün kullanılacağı ölçekleme stratejisi yukarıdan-aşağıda ise ölçekli iletişim örüntüsü yukarıdan-aşağıya, aşağıdan-yukarıya ise de iletişim örüntüsü aşağıdan-yukarıya doğru çarpanları kullanılarak ölçeklenir. Daha sonra her bölüme ait alt bölümler için de bu işlem özyineli olarak tekrarlanır.

Oluşturulan modelin ifade edilmesi için de Çizelge 8'de verilen mantıksal konfigürasyon bakış açısı tanımlanmıştır. Bu bakış açısı ile oluşturulan mantıksal konfigürasyon modelleri işlem birimleri ve aralarındaki iletişim ilişkileri ile gösterilebilmektedir.

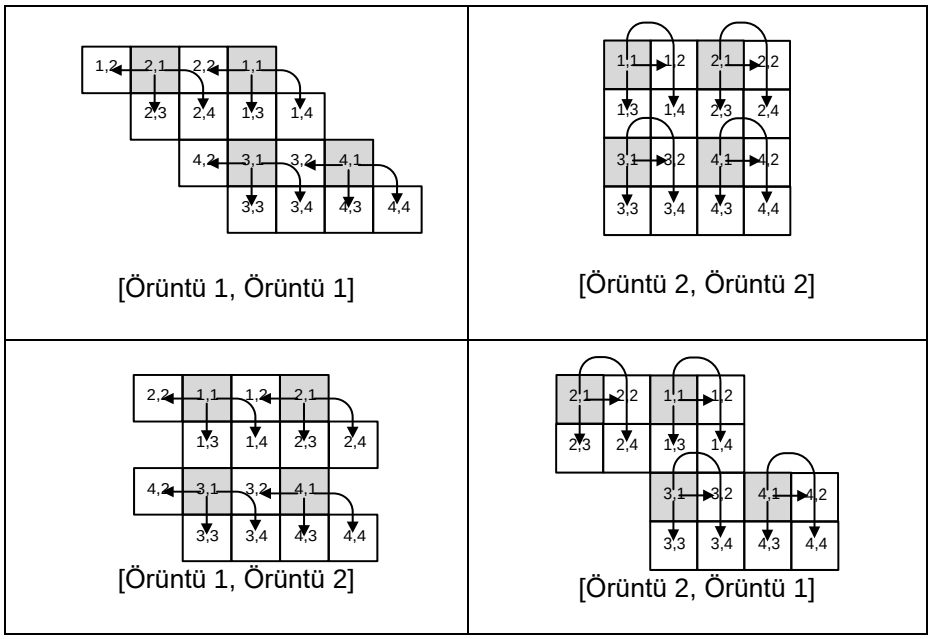
```
Procedure GenerateModel (section, size)
    if typeof(section) = SER then
        Add section with code
    endif
    if typeof(section) = PAR then
        Get the operation pattern from PAML
        if scaling = UP then
            create pattern bottom up
        endif
        if scaling = DOWN then
            create pattern top down
        endif
        ScalePattern(pattern, size)
    endif
    for each subsection in section
        GenerateModel(subsection)
    endfor
end
```

Şekil 36 Mantıksal konfigürasyonun oluşturulması algoritması.

Çizelge 8 Mantıksal Konfigürasyon Bakış Açısı

Adı	Mantıksal Konfigürasyon Bakış Açısı
İlgisi	Algoritma ve Fiziksel konfigürasyona göre mantıksal konfigürasyonun modellenmesi
Paydaşları	Mantıksal konfigürasyon mimarı
Öğeleri	Core – işlem biriminin modeli Dominating Core – Diğer düğümler ile veri alışverişinden sorumlu hakim işlem birimi
İlişkileri	Core kullanarak daha büyük döşemeler oluşturulur. Döşemeler ve iletişimler ile mantıksal konfigürasyon belirlenir.
Kısıtları	Core sayısı fiziksel konfigürasyondaki işlem birimi sayısı ile aynı olmalıdır. Core numaralndırması fiziksel konfigürasyondaki işlem birimi numaralndırması ile uyumlu olmalıdır.
Gösterimi	

Birden fazla sayıda örüntü olması durumunda birçok farklı mantıksal konfigürasyon modelinin de oluşturulması mümkündür. Örneğin Şekil 37'de daha önce matris çarpma algoritması için iki farklı döşeme ve iletişim örüntüsü kullanılarak oluşturulmuş mantıksal konfigürasyon seçenekleri verilmiştir. Bu mantıksal konfigürasyonlar iki farklı örüntü ile ölçeklemek için permutasyonları olan dört farklı ölçekleme sıralaması ile oluşturulmuştur.



Şekil 37 4x4 Fiziksel konfigürasyon için iki farklı iletişim örüntüsü kullanılarak oluşturulmuş mantıksal konfigürasyonlar.

4.2.2.5. Uygun Mantıksal Konfigürasyonları Seç

Farklı mantıksal konfigürasyon modelleri oluşturulduktan sonra bu modellerden uygun olanlarının seçilmesi ve bu modellerden kod üretiminin yapılması gerekmektedir. Bu seçimin yapılmasında çeşitli metrikler kullanılabilir. Ancak en temelde koşut algoritmalar için hızlanma (*speedup*) ve etkinlik (*efficiency*) metrikleri ön plana çıkmaktadır. Bu değerlerin koşut işlem performansına olan etkisi ise oluşan gecikme ile belirlenir (T). Oluşturulan mantıksal konfigürasyon modellerinde de bu değerler için bir hesaplama yöntemi uygulanabilir. Bunun için aşağıdaki formüller [29] kullanılmıştır:

$$T = \mathrm{R}(\alpha + \mathrm{L}\tau) \qquad (1)$$

R iletişim örüntüsü sayısı (iletişim tekrarlama sayısı)

α mesaj oluşturma maliyeti

L iletişim uzunluğu

τ iletişimdeki iletim maliyeti

Modellerde her mantıksal konfigürasyon için iletişim örüntülerinde gerçekleşen iletişim uzunlukları hesaplanmaktadır. İletişimin tekrarlanma sayısı olan iletişim örüntüsü sayısında $R\alpha$ mesajların toplam iletişim maliyetleri hesap edilir. $R\mathrm{L}$ burada toplam iletişim sayısını vermekte ve her iletişimdeli iletim maliyetinden toplam maliyet hesaplanabilir. Sonuçta çıkan gecikme değeri düşük olan modeller, uygun mantıksal konfigürasyon adayları olarak seçilebilir ve kod üretimi sağlanabilir.

4.2.2.6. Kodu Üret ve Konuşlandır

Yaklaşımdaki son adım ise seçilen uygun mantıksal konfigürasyon modelinden kodun üretilmesi ve bu kodun koşut hesaplama platformuna konuşlandırılarak çalıştırılmasıdır. Kodun konuşlandırılması ve çalıştırılmasında elle ya da otomatik yöntemler kullanılabilmekte [43][44], ancak bu konu bu tez kapsamında detaylı olarak incelenmemektedir.

Kodun üretilmesi için hedeflenen koşut hesaplama platformunda çalışan bir sistem kütüphanesinin seçilmesi gerekmektedir. Pratikte koşut hesaplama platformları için geliştirilmiş MPI, OpenMP, MPL, CILK gibi kütüphaneler bulunmaktadır [8].Koşut hesaplama platformunun özelliklerine göre bu kütüphanelerden birisi seçilebilir. Örneğin dağıtık mimarili bir koşut hesaplama platformu için MPI kullanmak gerekirken, paylaşımlı bellekli bir koşut heaplama platformu için OpenMP kullanmak daha uygun olabilir.

Kodun üretilmesi aşamasında model-güdümlü dönüştürme teknikleri kullanmak, bu çeşitliliği daha kolay yönetebilmemizi sağlar. Oluşan platformdan bağımsız model, seçilen platforma bağlı olarak tanımlanan dönüşüm kuralları ile platforma bağımlı modele ve oradan koda dönüştürülebilir. Bunların yapılabilmesi için öncelikle sistem kütüphanesine uygun sistem üst modelinin tanımlanması gerekir. Bu üst model ile sistem modelinin oluşturulması sağlanır ve son olarak da kod üretimi gerçekleştirilir. Bu kesimde bu aşamalar detaylı olarak anlatılmaktadır.

4.2.2.6.1 Sistem Üst Modeli

Koşut modelimizi oluşturduktan sonra kod oluşturmaya kadar geçen süreçte öncelikle hedef sistemde hangi amaca yönelik kod üretilmesi gerektiğinin belirlenmesi gerekir. MPI, OpenMP gibi yazılım çatılarını kullanarak çalışacak bir uygulama geliştirilebileceği gibi belli zamanlarda belli işlem birimlerinin,

anahtarlayıcıların davranışlarını üreten kodlar da üretilebilir. Model güdümlü yazılım geliştirmenin burada sağladığı en önemli fayda alan uzmanının oluşturduğu tek bir koşut modelden ihtiyaca göre tüm bu amaca yönelik model ve kodların üretilebilmesidir.

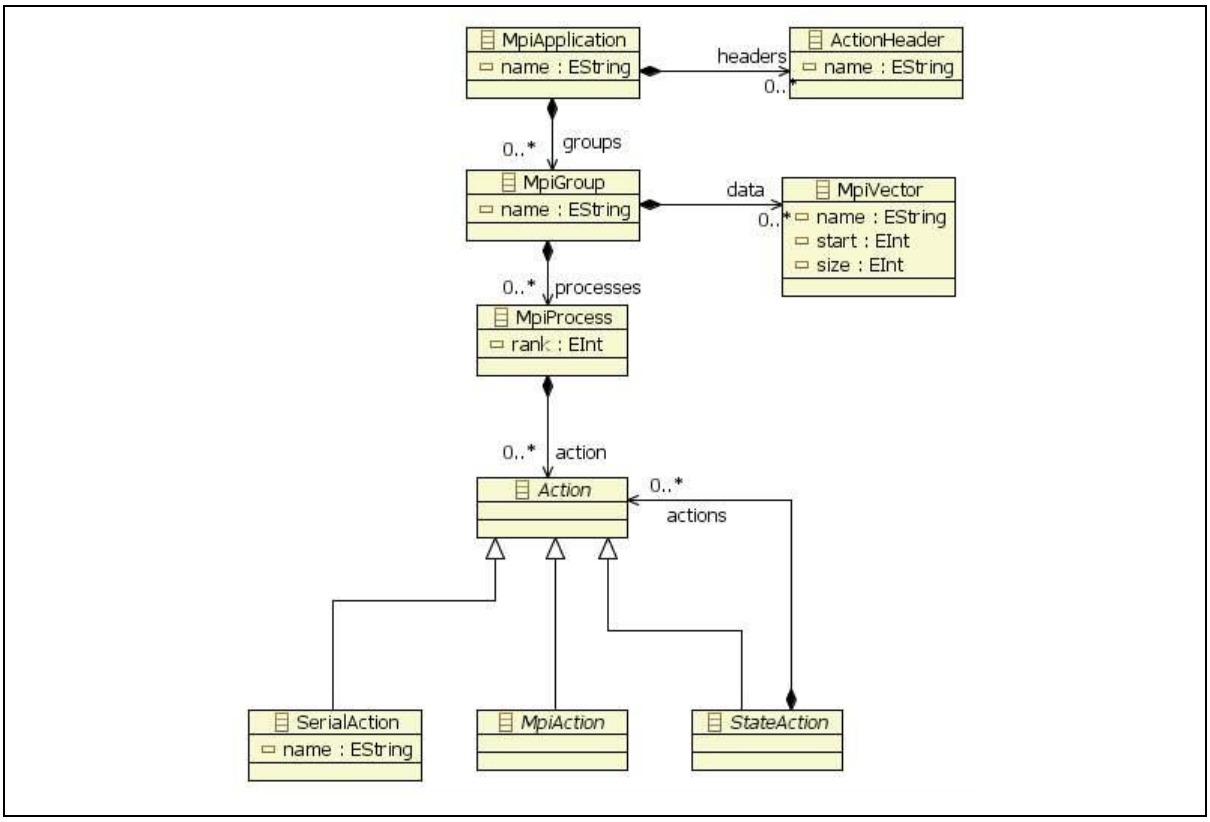
Bu tez kapsamında platforma özel model olarak iki örnek üzerinde odaklanılmıştır. Bu örnekler MPI yazılım çatısı kullanan bir uygulamanın kaynak kodunun üretilmesi ve donanım olarak sistem üzerinde bulunan anahtarlayıcı donanımların davranışlarının üretilmesidir.

4.2.2.6.1.1 MPI Üst Modeli

MPI için bir model güdümlü yazılım geliştirme ortamı oluşturmak için MPI üst modelinin oluşturulması gerekmektedir. MPI üst modeli bu alana özgü dil tanımlamalarının modellenebilmesine olanak verecek biçimde oluşturulması gerekmektedir.

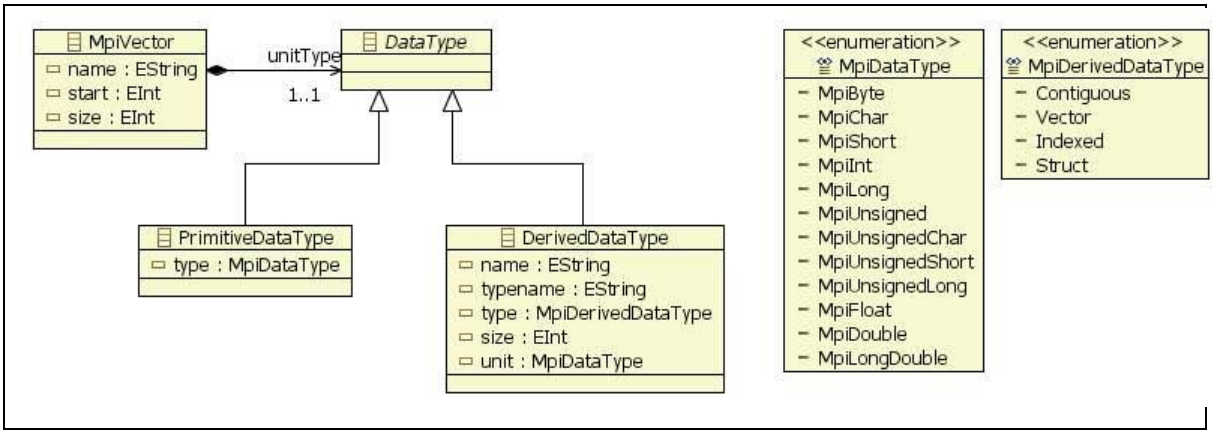
Temel bir MPI uygulaması işlemlerin birbirleriyle iletişim kurabileceğinin tanımlandığı haberleşme (*communicator*) ve grup (*group*) nesnelerini kullanır [45]. Çoğu MPI yordamı haberleşme nesnesini bir değişken olarak belirtmenizi ister. Her haberleşme içinde tüm işlemlerin sistem tarafından atanan kendi benzersiz tamsayı kimliği bulunur. İşlemler de diğer işlemlerden bağımsız olarak gerçekleştireceği işlem maddeleri (*action*) koşturmalıdır. Şekil 38'de bir MPI uygulamasının üst model içinde gösterimi verilmiştir.

Sistem modeli olarak MPI kullanan bir model üretilmesi için MPI bileşenlerini içeren bir MPI üst modelinin oluşturulması gerekmektedir. MPI için genel yapı ele alındığında her MPI uygulaması gruplardan (*groups*) oluşmuştur. Gruplar ise işlem birimlerini (*processes*) içermektedir.



Şekil 38 MPI üstmodeli.

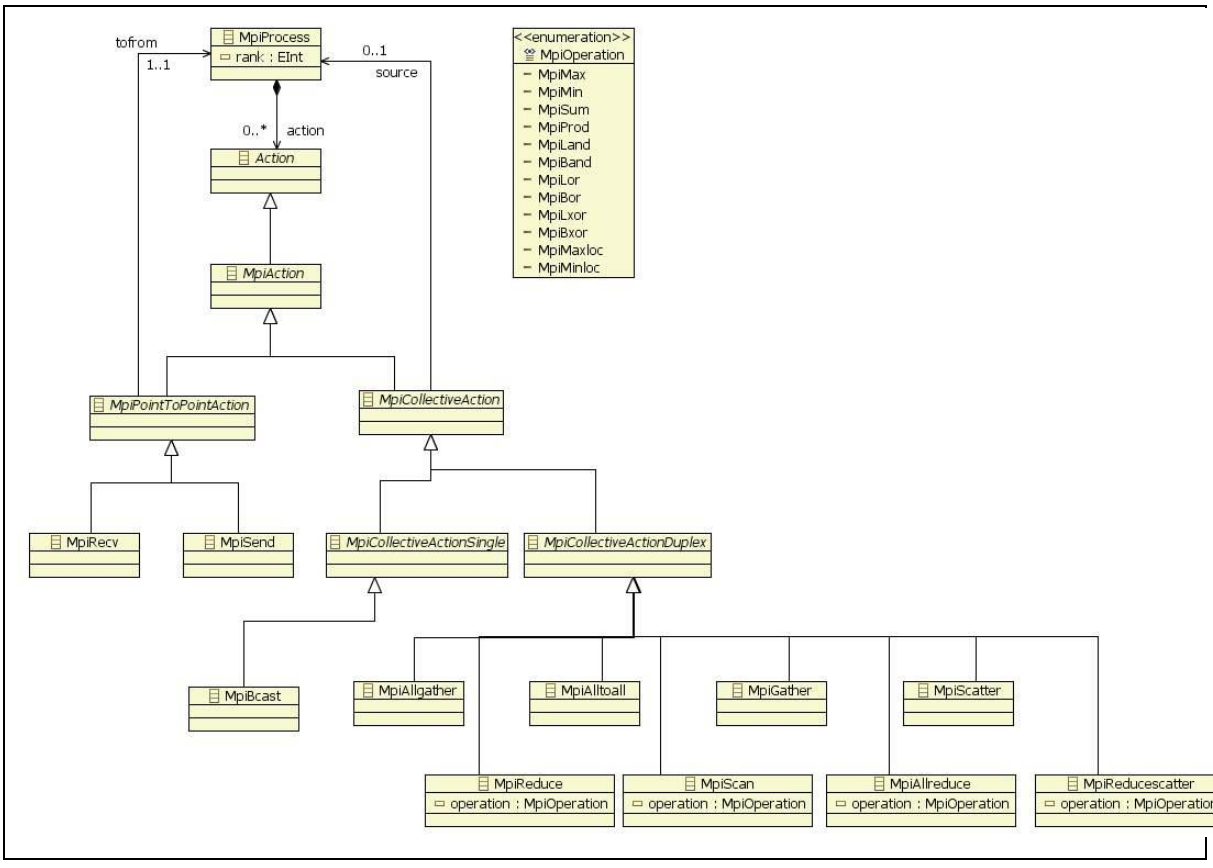
MPI içinde işlem birimlerinin yapacakları işlem maddeleri ise genel olarak ikiye akrılmaktadır. Bunlar seri işlemler (*SerialActions*) ve koşut çalışan MPI işlemleridir (*MPIAction*). Bunların dışında durum kontrollerinin gerçekleştirilmesi için durum işlemleri (*StateAction*) bulunur. Bu durum işlemleri if, for, while gibi durum kontrollerini içerecek olan seri işlemler olarak da değerlendirilebilir. Ancak bu durum kontrolleri seri ve koşut işlemleri de içerecek özyineli yapıda olması gerekmektedir. Bu bileşenleri içeren üst model ise Şekil 38'da verilmiştir.



Şekil 39 MPI veri üstmodeli.

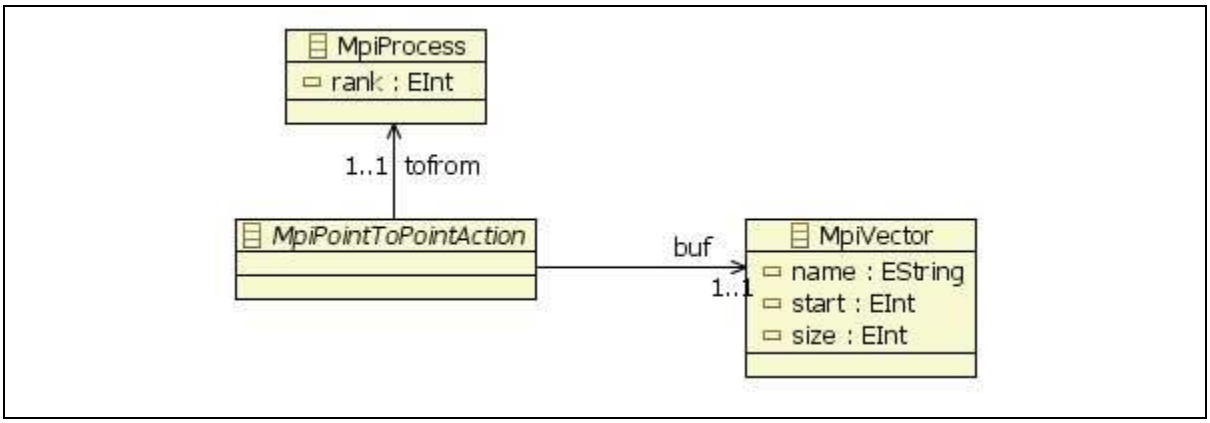
Belirlenen grup içinde hangi veri vektörü üzerinde işlemlerin yapılacağı belirlenmeli ve model içinde belirtilmelidir. Verilerin modellenmesi (Şekil 39), veri vektörünün birim elemanının tipinin ne şekilde olacağı ile belirlenir. Birim tipi ilkel (*primitive*) MPI tiplerinden ya da türetilmiş (*derived*) MPI tiplerinden birisi olabilir.

Koşut işlemde kullanılmak üzere MPI işlem maddeleri ise MPI alanına özgü biçimde noktadan noktaya (*point-to-point*) ya da toplu (*collective*) işlem maddelerinden oluşmaktadır (Şekil 40). Noktadan noktaya işlemlerde alma (*receive*) ya da gönderme (*send*) işlemleri seçilebilir. Bu işlemlerde seçilen işleme göre gönderilecek ya da alınacak (*tofrom*) MPI işlemi modelde seçilmektedir. Toplu işlemlerde ise bu işlemi gerçekleşmesini sağlayacak kaynak MPI işlemi seçilmelidir. Toplu işlemler tek yönlü (*single*) ya da çift yönlü (*duplex*) biçimde modellenebilmektedir.

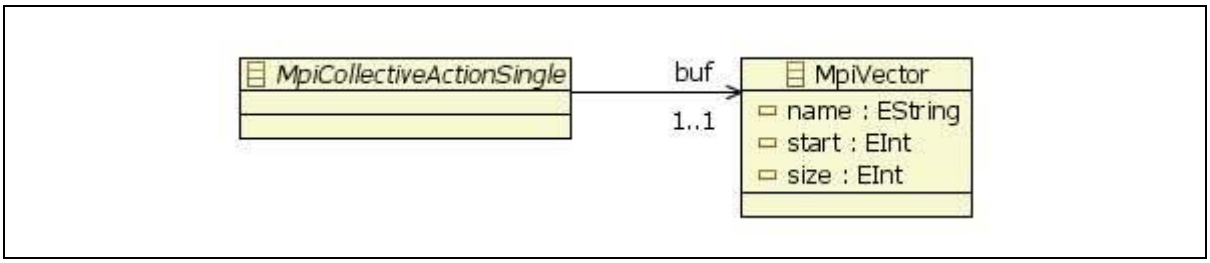


Şekil 40 MPI işlem maddeleri üstmodeli.

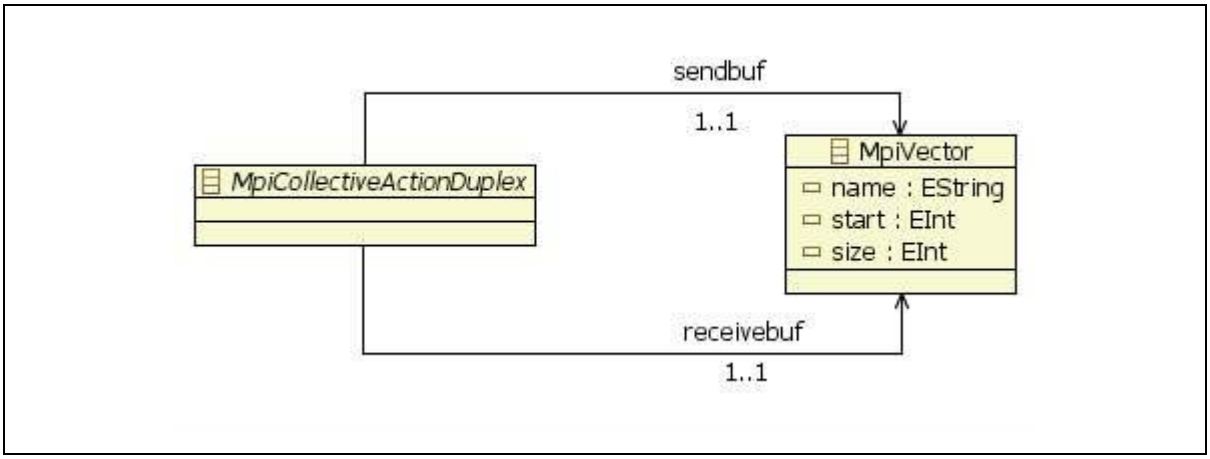
Noktadan noktaya işlemde (Şekil 41) ve tek yönlü toplu işlemde (Şekil 42) tek bir tampon (*buffer*) veri vektörü kullanılmaktadır. Çift yönlü toplu işlemde (Şekil 43) ise hem gönderilen hem de alınan tampon veri vektörleri tanımlanmalıdır.



Şekil 41 MPI noktadan noktaya işlem üstmodeli.

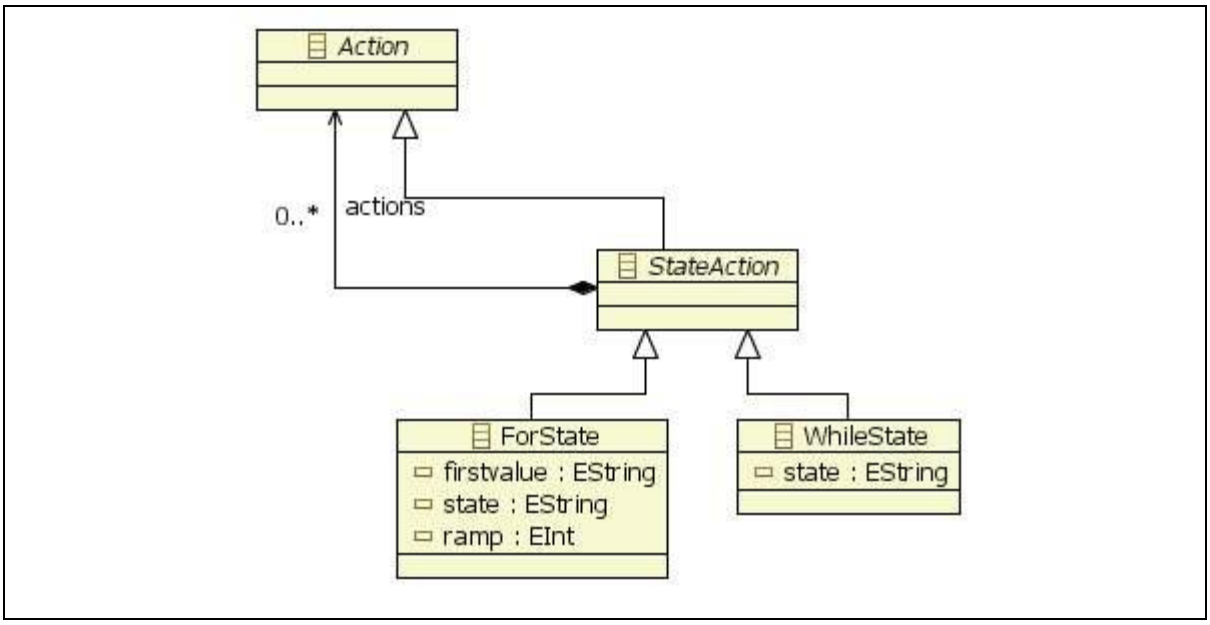


Şekil 42 MPI tek yönlü toplu işlem üstmodeli.



Şekil 43 MPI çift yönlü toplu işlem üstmodeli.

MPI işlem maddelerinin işlemler içinde modellenmesinin yanında bazı durumlarda programlama dillerine has durum kontrol yapılarının da modellenmesi gerekebilir. Durum (*state*) işlem maddeleri için *for* ve *while* durum kontrolleri üst modele eklenmiştir (Şekil 44).

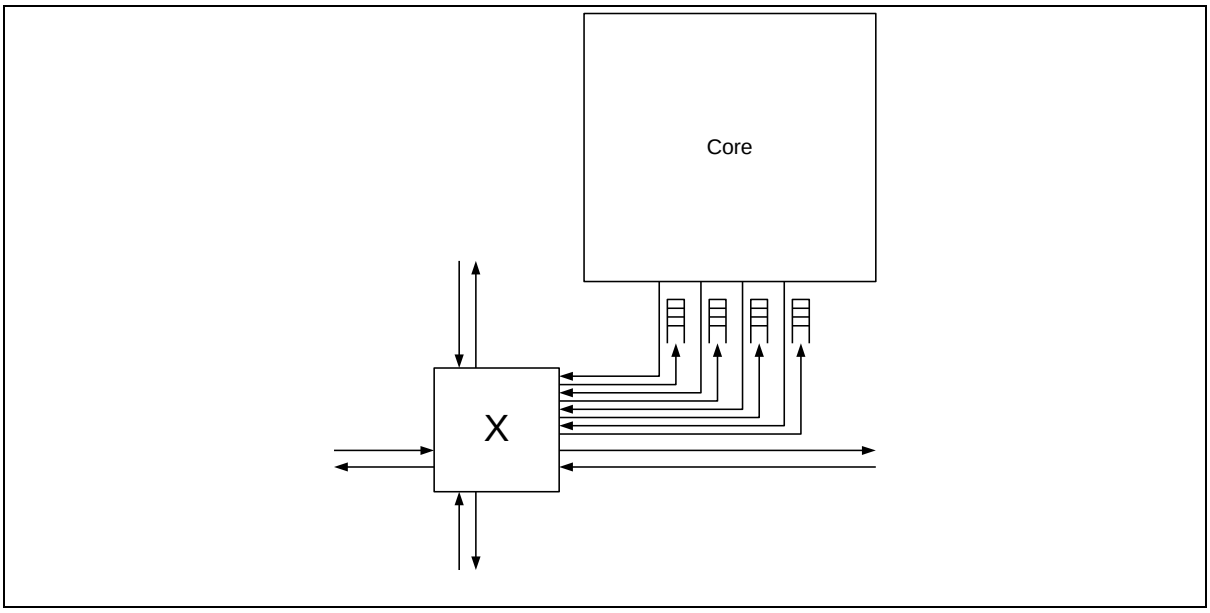


Şekil 44 MPI durum (*state*) işlemleri üstmodeli.

4.2.2.6.1.2 NoC Üst Modeli

Bir önceki kesimde anlatılan MPI üst modeli, geliştirilen koşut modelin çalışacak olan koşut sistemdeki yazılım kodlarının üretilmesi için kullanılacaktır. Ancak yazılım kodlarının otomatik üretilmesinin yanında, tezde belirtilen model güdümlü yaklaşımın farklı kullanımları da mümkün olmaktadır. Bu kesimde donanım tarafına bakarak, donanımın yazılım ile birlikte ne şekilde yapılandırılabileceğinin üzerinde durabiliriz.

Bu kesim için yongaya bağlı ağ (network on chip - NoC) örneği anlatılacaktır. NoC yongaları bir işlemci birimine bağlı, ve bu işlemci birimi tarafından ağ durumları ve yönlendirmeleri kontrol edilebilen yongalardır [46]. Şekil 45'de NoC yongalarının kavramsal gösterimi yer almaktadır.



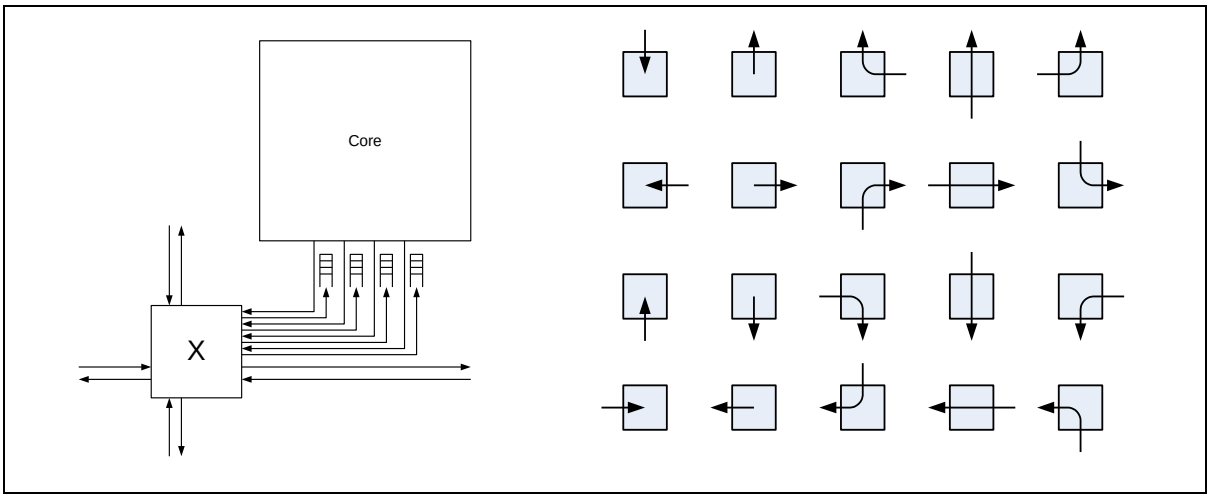
Şekil 45 Yongaya bağlı ağ bileşeni kavramsal gösterimi.

Bu yongalar çalıştırılacak uygulamanın ağ üzerinde iletişim yapısına bağlı olarak ayarlanması gerekmektedir. Geliştirdiğimiz koşut modelde bu iletişimlerin ne şekilde yapılacağı iletişim örüntüleri ile tanımlanmıştı. Bu koşut iletişim örüntüleri kullanılarak bu yongaların ayarları belirlenebilir.

Bu yongaların iletişim örüntüleri ile kullanılması, öncelikle iletişimin hangi yönde ve hangi birimlerde yapıldığının bilinmesi sayesinde o yonga kullanılmadığında gücünün kesilmesi devingen olarak sağlanabilir. Bu durum enerji harcamasında eniyilemenin donanım seviyesinde yapılmasını sağlar. Ayrıca daha önceden iletişim örüntülerine uygun olarak ayarlanarak oluşacak gecikmelerin önüne geçilebilecektir.

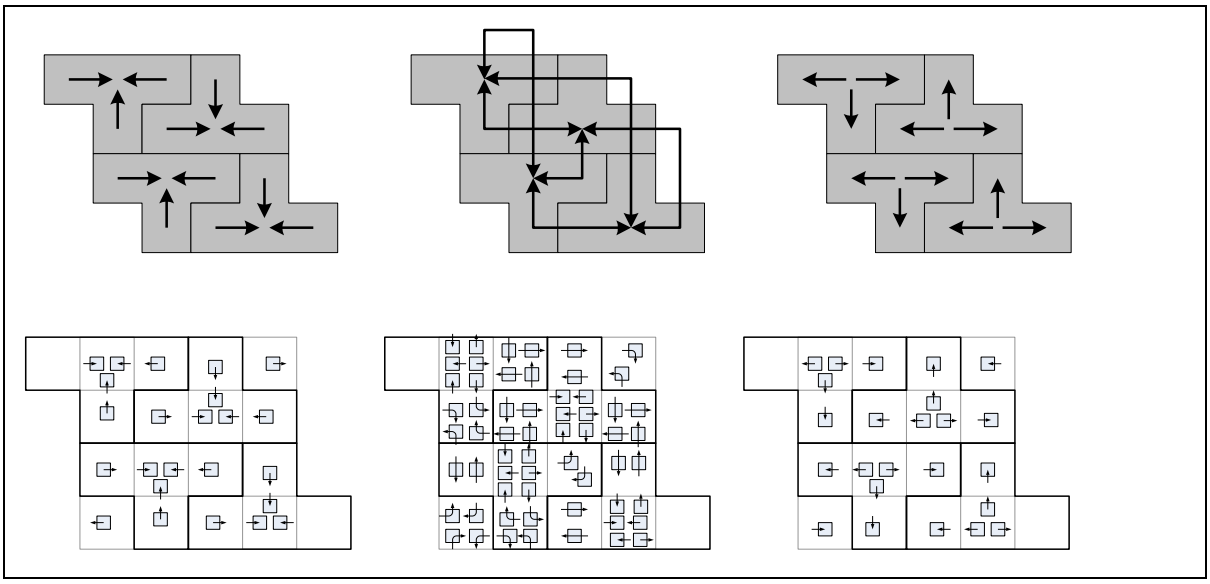
NoC yongalar üzerinde yapılabilecek olan temel ayarlar, gelen verinin işlemci birimine yönlendirme ya da işlemci biriminden ağa yönlendirme ve gelen veriyi kuzey, güney, doğu ya da batıya yönlendirme şeklinde sınıflandırılabilir. Şekil 46'de ilgili NoC yongası için bu ayarlar çizgesel olarak gösterilmiştir.

Bu temel ayarların koşut modelimizden elde edilmesi de yine oluşturduğumuz döşemeler ve iletişim örüntülerine göre yapılacaktır. Yine tüm-değişim algoritması üzerinden gidecek olursak (Şekil 47), ilk adımda her bir küçük örüntüde yönetilen düğümler hakim düğüme verilerini göndermektedir.



Şekil 46 NoC yongalar için temel ayarlar.

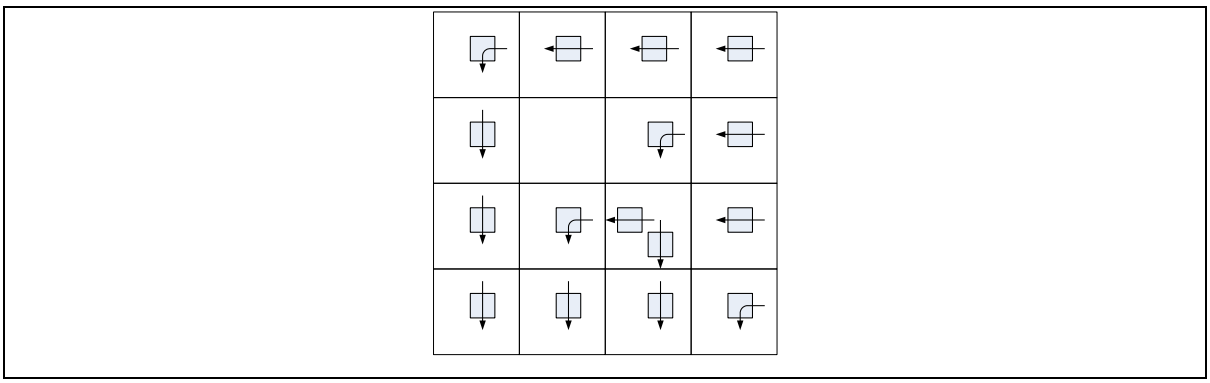
Sol üst köşede bulunan örüntü düşünüldüğünde yönetilen düğümler kendinden batıya, kendinden doğuya ve kendinden kuzeye veri iletirken hakim düğüm doğudan kendine, batıdan kendine ve güneyden kendine veri almaktadır. Bu durumda ilgili NoC yongalarında bu iletişim örüntüsüne uygun veri iletişim ayarının yapılması gerekecektir. Tüm iletişim örüntüsü bu şekilde işletildiğinde istenilen ayarların Şekil 47'de gösterildiği biçimde elde edilmesi gerekecektir.



Şekil 47 Tüm-değişim algoritması için NoC durumlarının gösterimi.

Bir düğümde elde edilen ayarlar ise özyineli ya da tekrarlayıcı biçimde parçalandığında ise daha alt düzeylerde ne şekilde belirleneceği de oluşturulmalıdır. Bu noktada hakim düğümler kilit rol oynar. Örnek olarak ele aldığımız tüm-değişim algoritmasında bir düğüm özyineli parçalandığı düşünüldüğünde hakim düğüme

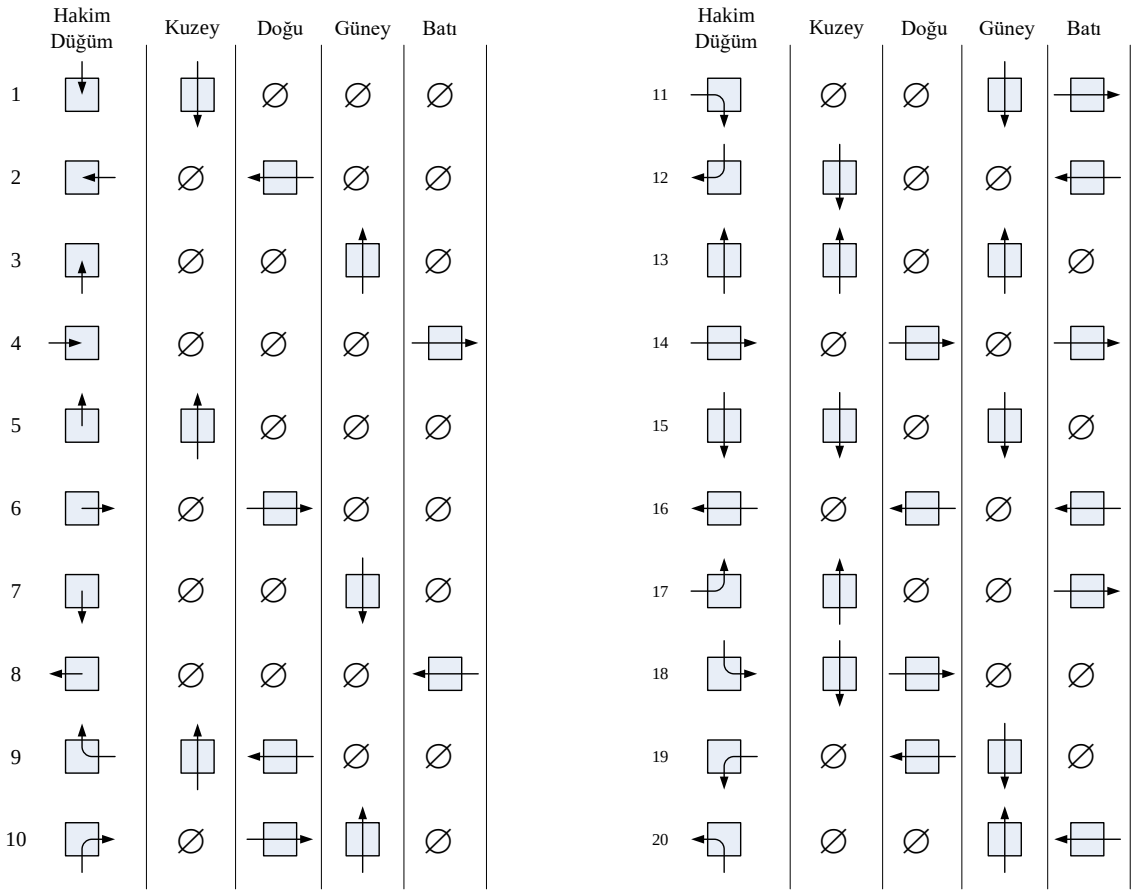
göre tüm düğümlerin ayarları bulunacaktır. Şekil 48'de doğudan güneye bir yönlendirme için örnek verilmiştir. Doğu tarafında bulunan tüm düğümler aynen batıya veriyi batıya geçirirken en alttaki hakim düğüm doğudan güneye veriyi çevirmektedir. Her satırda hakim düğüme gelene kadar veri batıya geçirilir, hakim düğüme gelince de güneye yönlendirilir. Güneye yönlendirilen veri de o sütun boyunca kuzeyden güneye yönlendirilerek devam eder. Bu sayede güney tarafından tüm veriler çıkış yapar.



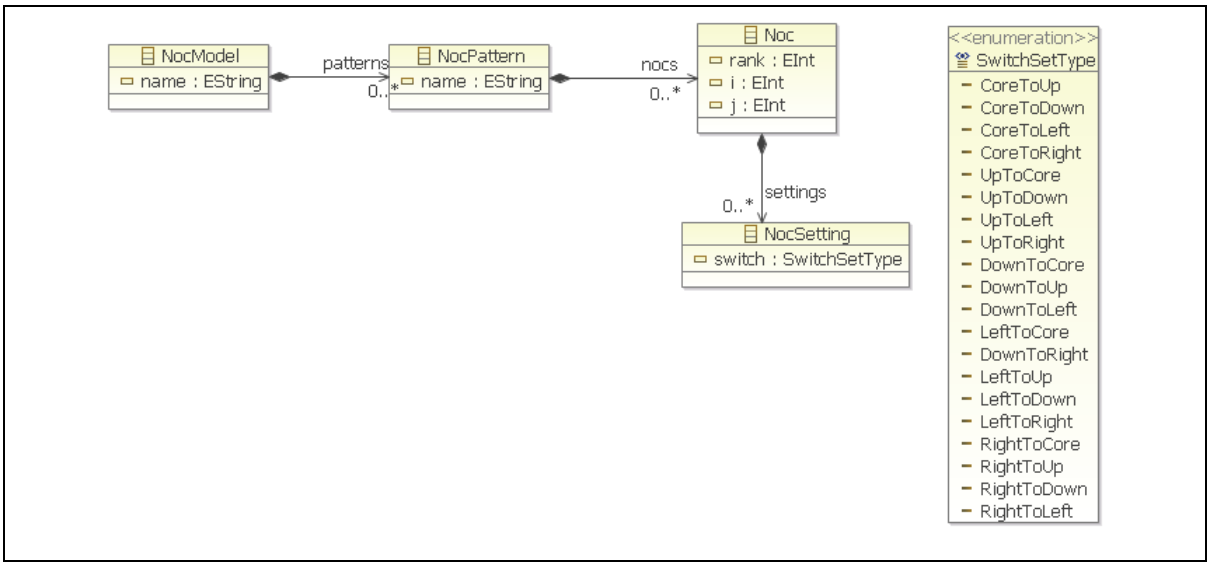
Şekil 48 Düğümün özyineli parçalanması sonucu oluşan alt durumların gösterimi. Doğudan güneye bir durumun alt durumları gösterilmektedir.

Bu kurallar incelendiğinde hakim düğümün komşularında NoC yongalarının veri iletişimi ile ilgili hakim düğüme göre davranış belirlediği görülmektedir. Ana davranış doğudan güneye olduğunda alt düzeydeki döşemelerde hakim düğümlerin kuzey ve batı komşularında bir ayar gerekmezken doğu ve güney komşularında bu iletişime uygun olarak doğudan batıya ve kuzeyden güneye iletişimi gerçekleştirilir. Bu davranışlar her genel davranış için sınıflandırıldığında Çizelge 3'de verilen davranışlar elde edilmektedir.

Çizelge 9 Alt düğümdeki hakim düğüme gore komşularının durumları çizelgesi.



Çizelge 3'de verilen sınıflandırma bizim dönüşümde hangi kuralların işletileceğini göstermektedir. Bu sebeple bu dönüşüm ile amaçladığımız davranışları belirleyen üst modelin belirlenmesi sağlanabilir. Bir koşut modelden oluşturulacak olan NoC modelinin içinde NoC örüntüleri (*NoCPattern*) biçiminde ayrıştırılması sağlanmalıdır. Her örüntü de bir NoC yongasına ait bilgileri (NoC) ve ona ait birden fazla ayarı (*NoCSetting*) içermesi gerekecektir. Bu ayarlar da anahtarlama durumları (*SwitchSetType*) ile belirlenir. İlgili tanımlamalarla oluşturulan üst model Şekil 49'de verilmiştir.



Şekil 49 NoC üstmodeli.

4.2.2.6.2 Sistem Modelinin Oluşturulması

Sistem üst modelin belirlenmesinden sonra geliştirilen koşut modelin sistem modeline dönüştürülmesi işlemi gerçekleştirilecektir. Bu amaçla geliştirilen koşut üst modeli ve bir önceki kesimde tanımlanan sistem üst modeli arasında dönüşüm kurallarının tanımlanması gerekmektedir. Bu dönüşüm kuralları kullanılarak geliştirilen koşut model sistem modeline otomatik olarak dönüştürülebilir.

Sistem modelinin oluşturulması için modelden-modele dönüşüm teknikleri kullanılacaktır. Modelden-modele dönüşüm için ATL [47] dili dönüşüm motoru olarak kullanılmıştır. MPI modelinin ve NoC modelinin oluşturulabilmesi için iki farklı dönüşüm kuralları kümesi oluşturulmuştur. Bu kesimde örneklediğimiz bu iki farklı sistem modelinin oluşturulması anlatılmaktadır.

4.2.2.6.2.1 MPI Modelinin Oluşturulması

Daha önceki kesimlerde belirtildiği gibi modelden-modele dönüşümde gerçekleştirilecek dönüşüm kurallarının tanımlanması için bir dönüşüm motorunun kullanılması gerekmektedir. Bu tez kapsamında verilen örneklemelerde ATL dönüşüm dili ve motoru kullanılarak bu dönüşüm gerçekleştirilmiştir.

Bir önceki kesimde koşut modelden dönüştürülmesini sağlamak üzere iki farklı üst model tanımlamıştık. Bu kesimde koşut modelimizin bu iki üst modelden birisi olan MPI üst modelini kullanarak MPI modeline dönüştüreceğiz.

Koşut üst modelimizde ana bileşenimiz ParallelModel bileşenimizdi. Bu bileşenin dönüşümlerde MPI üst modelindeki MpiApplication bileşenine dönüştürmek üzere kurallarımızı tanımlamamız gerekmektedir (Şekil 50). Burada ayrıca işlem birimlerinin de MpiGroup bileşeni altında birleştirilmesi gerekmektedir. Tüm işlem birimleri de MPI ile kullanılacak biçimde liste halinde dönüştürülecektir.

```
rule model2application {
      from
            mdl : parallel!ParallelModel
      to
            app : mpimetamodel!MpiApplication (
                        name <- mdl.name, groups <- grp),
            grp : mpimetamodel!MpiGroup (
                        name <- mdl.name, processed <- coreList),
            coreList : distinct parallel!Core foreach(core in_
                        mdl.pattern.getAllCores())
                              ( rank <- core.Rank )
}
```

Şekil 50 Koşut modelden MPI modeline dönüşüm kuralları.

Bu dönüşüm kuralları tanımlandığında kaynak modeldeki alt bileşenlere erişim sağlanarak uygun biçime getirilip hedef modele eklenmesi gerekmektedir. Bu noktada yardımcı bağlam (*helper context*) dönüşümlerinin yazılması faydalı olmaktadır (Şekil 51). Koşut modelimizin özyineli ya da tekrarlayıcı olarak alt parçalara ayrılmış olduğu düşünüldüğünde bu yardımcı bağlamlarında özyineli biçimde alt bileşenlere erişimi sağlanmalıdır.

```
helper context parallel!Pattern def : getAllCores() :
OrderedSet(parallel!Core) =
      self.children->iterate( child ; elements :
OrderedSet(parallel!Node) =
            OrderedSet{} | if child.oclIsTypeOf(parallel!Pattern) then
                              elements.append(child.getAllCores())
                        else
                              elements.append(child)
                        endif);
```

Şekil 51 Alt parçaların oluşturulması için yardımcı bağlam.

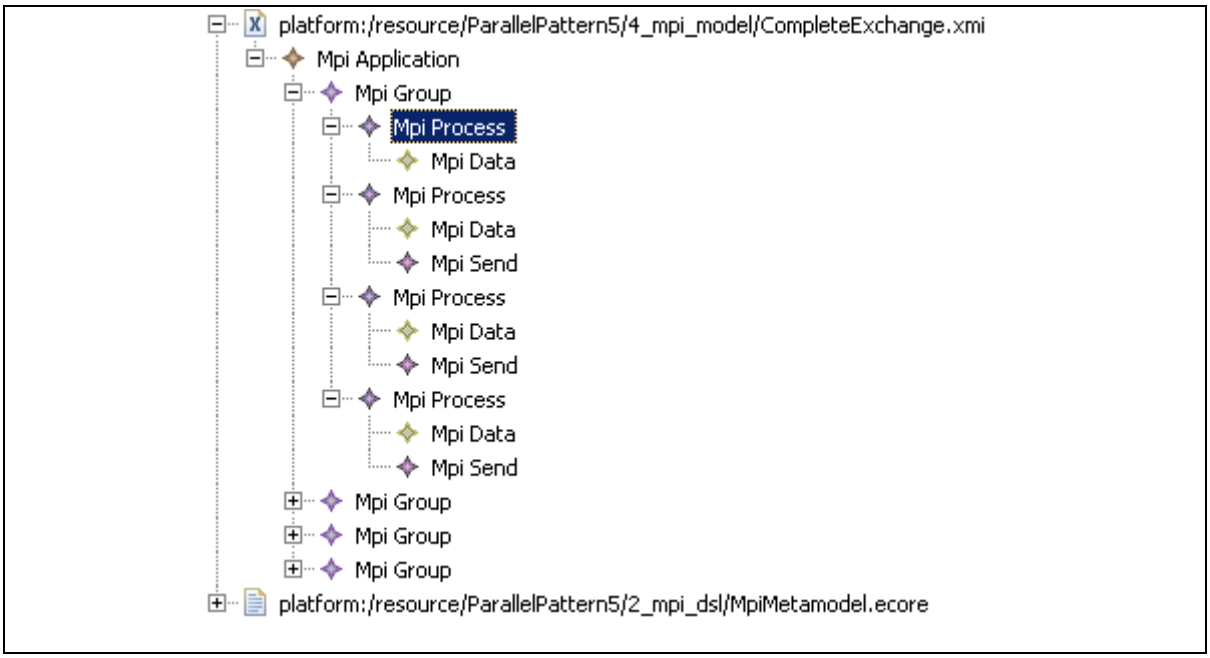
Bu özyineli erişim sağlanırken de düğümlerin en son işlem birimleri mi yoksa ara düğüm olan örüntüler mi olduğunun kontrol edilmesi gerekmektedir. Bu işlem için ATL için tanımlı temel tip kontrolleri yapılacaktır (Şekil 52).

```
helper context parallel!Node def : isCore() : Boolean =
      not self.refImmediateComposite().oclIsUndefined();
helper context parallel!Pattern def : getPatterns () :
OrderedSet(parallel!Pattern) =
      self.children->iterate( child ; elements :
OrderedSet(parallel!Pattern) =
            OrderedSet{} | if child.oclIsTypeOf(parallel!Pattern) then
                              elements.append(child)
                        else
                              OclUndefined
                        endif);
helper context parallel!Pattern def : getCores () :
OrderedSet(parallel!Core) =
      self.children->iterate( child ; elements :
OrderedSet(parallel!Core) =
            OrderedSet{} | if child.oclIsTypeOf(parallel!Core) then
                              elements.append(child)
                        else
                              OclUndefined
                        endif);
helper context parallel!Node def : getDominatingCore() : parallel!Core =
      if self.oclIsTypeOf(paralel!Pattern) then
            self.dominating.getDominatingCore()
      else
            self
      endif;
```

Şekil 52 Son işlem birimi kontrolü yapan ve tüm işlem birimlerini bulan yardımcı bağlam.

Bu dönüşüm kuralları tanımlandıktan sonra iletişim metodlarının ve yapılacak faaliyetlerin dönüşümü gerçekleştirilecektir. Burada en alt düzeyde işlem biriminden işlem birimine iletişimler aynen geçirilirken, örüntü düğümden örüntü düğümlere olan iletişimler, o örüntülere ait hakim düğümler arasında olacaktır. Bunun için de özyineli olarak bir örüntünün en alt düzeydeki hakim işlem birimi bulunmaktadır.

Gerçekleştirilen dönüşüm kuralları ATL dönüşüm motoru kullanılarak çalıştırıldığında Şekil 53'de verilen MPI modeli oluşturulmaktadır. Bu model bir sonraki aşama olan kod oluşturma safhasına hazır hale gelmektedir.



Şekil 53 Tüm-değişim örneği için dönüştürülen MPI modeli.

Her ne kadar oluşturulan MPI modeli kod dönüşümüne elverişli olsa da, çalıştırılması amaçlanan son amaç kodu karşılamayabilir. Model-güdümlü yazılım geliştirmenin doğasında olan ve bir aşamada elle değiştirme gerektirebilen bu yöntemde, yapılacak olan elle değiştirmeler istenirse bu aşamada yapılabilir, ya da kod dönüşümünden sonra kod üzerinde de yapılabilir.

4.2.2.6.2.2 NoC Modelinin Oluşturulması

MPI modelinin oluşturulmasından sonra donanım tarafında yapılacak olan yapılandırmanın oluşturulacağı NoC modelinin oluşturulması sağlanmalıdır. ATL dili kullanılarak modelden model dönüşüm için dönüşüm kurallarının yazılması gerekmektedir. Koşut modelimizin NoC modeline dönüşümü için temel olarak tüm örüntülerin listeli olarak elde edilmesi ve daha sonra koşut örüntülerimizin NoC örüntülerine, tanımlı düğümlerimizin NoC yapılandırma ayarlarına ve işlem birimlerimizin de NoC yapılarına dönüştürülmesi gerekecektir.

Bu dönüşümler için gereken kural tanımlamaları Şekil 54'de verilmiştir. Burada örüntülere ait tüm işlem birimlerinin alınması ve hakim düğümün bulunması gibi yardımcı kavram tanımlamaları, MPI dönüşümlerinde kullanılan kurallardan aynen alınmıştır.

```
rule model2noc {
      from mdl : ParallelModel!ParallelModel
      to nocmdl : NocModel!NocModel (
                  name <- mdl.name,
                  patterns <- mdl.pattern->iterate(step; pat :
                              OrderedSet(ParallelModel!Pattern) =
OrderedSet{} |
                        if(step.oclIsTypeOf(ParallelModel!Pattern)) then
                              pat.append(step)
                        else
                              OclUndefined
                        endif))
}
rule pattern2nocpattern {
      from
            pat : ParallelModel!Pattern (pat.isStep())
      to
            nocpat : NocModel!NocPattern(
                  name <- pat.name,
                  nocs <- pat.getAllCores())
}
rule node2setting {
      from
            node : ParallelModel!Node
      to
            setting : NocModel!NocSetting(
                  switch <- node.getSettings())
}
rule core2noc {
      from
            core : ParallelModel!Core
      to
            noc : NocModel!Noc(
                  rank <- core.rank,
                  i <- core.i,
                  j <- core.j)
}
```

Şekil 54 Koşut modelden NOC modeline dönüşüm kuralları.

NoC yapılandırma ayarlarının elde edilmesinde ise, üst modelde tanımlanan yönlendirme durumlarının tanımlanması gerekmektedir. Bu amaçla sıralama (*enumeration*) yapısında yönleri belirten tanımlamalar yapılmıştır ve NoCSetting yapılandırma ayarlarında bu sıralama değerleri kullanılacaktır (Şekil 55).

```
helper def : CORE_TO_UP : OclAny = 1;
helper def : CORE_TO_DOWN : OclAny = 2;
helper def : CORE_TO_LEFT : OclAny = 3;
helper def : CORE_TO_RIGHT : OclAny = 4;
helper def : UP_TO_CORE : OclAny = 5;
helper def : UP_TO_DOWN : OclAny = 6;
helper def : UP_TO_LEFT : OclAny = 7;
helper def : UP_TO_RIGHT : OclAny = 8;
helper def : DOWN_TO_CORE : OclAny = 9;
helper def : DOWN_TO_UP : OclAny = 10;
helper def : DOWN_TO_LEFT : OclAny = 11;
helper def : DOWN_TO_RIGHT : OclAny = 12;
helper def : LEFT_TO_CORE : OclAny = 13;
helper def : LEFT_TO_UP : OclAny = 14;
helper def : LEFT_TO_DOWN : OclAny = 15;
helper def : LEFT_TO_RIGHT : OclAny = 16;
helper def : RIGHT_TO_CORE : OclAny = 17;
helper def : RIGHT_TO_UP : OclAny = 18;
helper def : RIGHT_TO_DOWN : OclAny = 19;
helper def : RIGHT_TO_LEFT : OclAny = 20;

helper def : NORTH : OclAny = 31;
helper def : SOUTH : OclAny = 32;
helper def : EAST : OclAny = 33;
helper def : WEST : OclAny = 34;
```

Şekil 55 NOC sıralama tanımlamaları.

Bu aşamadan sonra bulunan hakim düğümlere göre her bir işlem birimine ait NoC için ayarların belirlenmesi gerekir. Hakim düğümün kuzeyinde, güneyinde, doğusunda ya da batısında bulunan ve üst model tanımlamasında açıklanan kural çizelgesine göre (Çizelge 9) her NoC için yapılandırma ayarları belirlenir (Şekil 56).

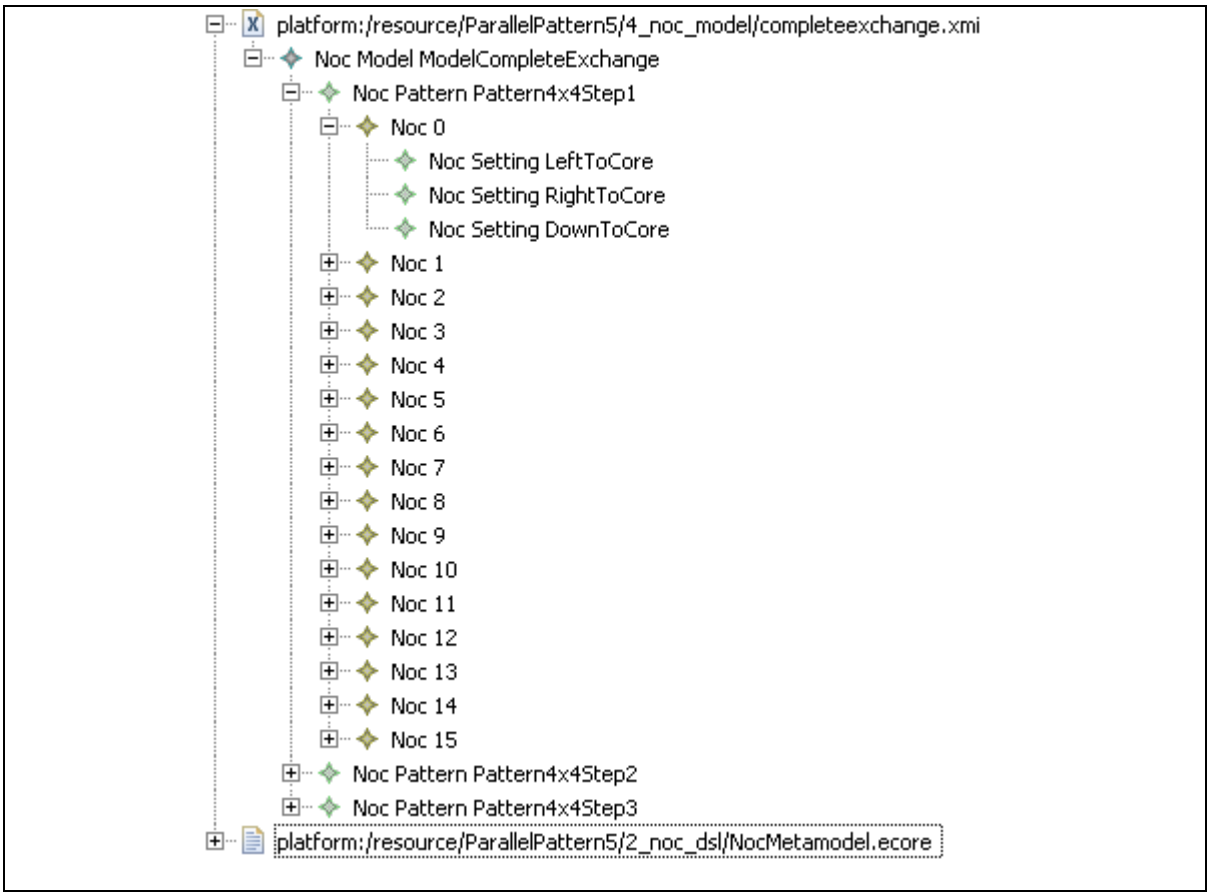
```
helper context ParallelModel!Node def : getNocSettings(comm : String) :
OclAny = OclUndefined;
helper context ParallelModel!Node def : getSettings(setting : OclAny) :
      OrderedSet(NocModel!NocSetting) =
      let parent : ParallelModel!Pattern = self.refImmediateComposite()
in
      if parent.oclIsTypeOf(ParallelModel!Pattern) then
      parent.communications->iterate(comm; set :
OrderedSet(NocModel!NocSetting) = OrderedSet{} |if comm.from = self then
      if (comm.to.getDominatingCore().i -
      self.getDominatingCore().i).abs() > (comm.to.getDominatingCore().j
- self.getDominatingCore().j).abs() then
      if (comm.to.getDominatingCore().i -
      self.getDominatingCore().i) > 0 then
      --NORTH
      if self.isCore() then
            set.append(
      thisModule.convertSetting(NORTH, setting) )
      else
            self.nodes->iterate(node; e: OclAny =
                  OclUndefined |    node.getSetting(setting))
      endif
      else
      --SOUTH
      if self.isCore() then
            set.append(
      thisModule.convertSetting(SOUTH, setting) )
      else
            self.nodes->iterate(node; e: OclAny =
                  OclUndefined | node.getSetting(setting))
      endif
      endif
      else
      if (comm.to.getDominatingCore().j - self.getDominatingCore().j) > 0
      then
      --EAST
      if self.isCore() then
            set.append( thisModule.convertSetting(EAST,setting) )
      else
            self.nodes->iterate(node; e: OclAny = OclUndefined |
            node.getSetting(setting))
      endif
      else
      --WEST
      if self.isCore() then
            set.append( thisModule.convertSetting(WEST, setting) )
      else
            self.nodes->iterate(node; e: OclAny = OclUndefined |
            node.getSetting(setting))
      endif
      endif
      endif
      else
            OclUndefined
      endif)
      else
            OclUndefined
      endif;
```

Şekil 56 Hakim düğümün pozisyonuna göre NOC ayarlarını oluşturan yardımcı bağlam.

Bu ayarlamaların belirlenmesinde yapılandırmanın hakim düğüme göre tersine şekilde dönüştürülmesi gerekir. Bu dönüşümün bir önceki dönüşüm içinde kullanılması sonucu son ayarlar elde edilmiş olacaktır (Şekil 57). Dönüşüm tamamlandığında elimizde NoC modeli Şekil 58'de gösterildiği gibi otomatik olarak oluşturulmuş olur. Bu NoC modeli kullanılan donanım türüne göre dönüşüme uygun, yapılandırma ayarlarını içeren bir modeldir.

```
helper def : convertSetting(pos : OclAny, setting : OclAny) : OclAny =
      if setting = thisModule.UP_TO_CORE then
            if pos = thisModule.NORTH then
                  thisModule.UP_TO_DOWN
            else
                  OclUndefined
            endif
      else
            if setting = thisModule.RIGHT_TO_CORE then
                  if pos = thisModule.EAST then
                        thisModule.RIGHT_TO_LEFT
                  else
                        OclUndefined
                  endif
            else
                  if setting = thisModule.DOWN_TO_CORE then
                        if pos = thisModule.SOUTH then
                              thisModule.DOWN_TO_UP
                        else
                              OclUndefined
                        endif
                  else
                        if setting = thisModule.LEFT_TO_CORE then
                              if pos = thisModule.WEST then
                                    thisModule.LEFT_TO_RIGHT
                              else
                                    OclUndefined
                              endif
                        else
                              OclUndefined
                        endif
                  endif
            endif
      endif;
```

Şekil 57 NOC ayarlarının dönüşümü.



Şekil 58 Tüm-değişim örneği için dönüştürülen NoC modeli.

4.2.2.6.3 Kod Üretilmesi

4.2.2.6.3.1 MPI Kod Üretilmesi

MPI için model dönüşümünde kullanmak için gereken üst modelin tanımlanmasından sonra modelden metne dönüşüm için gereken şablonların oluşturulması gerekmektedir. Üst model içinde tanımlanan uygulama yapısı, verilerin modellenmesi ve koşut işlemler için kod üretim şablonlarının tanımlanması gerekmektedir. Yapılan çalışmada Xpand [26] aracı model dönüştürme motoru olarak kullanılmış ve bu araç için bu kod üretim şablonları oluşturulmuştur.

MPI grubu içinde tanımlanan verilerin kod içinde C/C++ veri tiplerine dönüştürülmesi gerekmektedir. Bunun için cTypeDefinition şablonu tanımlanmıştır (Şekil 59).

```
«DEFINE cTypeDefinition FOR MpiDataType»
	«IF this.toString().endsWith("MpiByte")»byte«ENDIF»
	«IF this.toString().endsWith("MpiChar")»char«ENDIF»
	«IF this.toString().endsWith("MpiShort")»short«ENDIF»
	«IF this.toString().endsWith("MpiInt")»int«ENDIF»
	«IF this.toString().endsWith("MpiLong")»long«ENDIF»
	«IF this.toString().endsWith("MpiUnsigned")»unsigned int«ENDIF»
	«IF this.toString().endsWith("MpiUnsignedChar")»unsigned
char«ENDIF»
	«IF this.toString().endsWith("MpiUnsignedShort")»unsigned short
	«ENDIF»
	«IF this.toString().endsWith("MpiUnsignedLong")»unsigned
long«ENDIF»
	«IF this.toString().endsWith("MpiFloat")»float«ENDIF»
	«IF this.toString().endsWith("MpiDouble")»double«ENDIF»
	«IF this.toString().endsWith("MpiLongDouble")»long double«ENDIF»
«ENDDEFINE»
```

Şekil 59 cTypeDefinition şablonu.

cTypeDefinition şablonu kod üretecinde ilkel tip bir veri tipi tanımlaması yapıldığında kullanılmaktadır. Türetilmiş verilerin de tanımlaması ile birlikte typeDefinition şablonunda (Şekil 60) ilkel tipler de tanımlanmaktadır.

```
«DEFINE typeDefinition FOR DataType»
	«IF this.metaType.name.endsWith("DerivedDataType")»
		«((DerivedDataType)this).typename»
	«ELSE»
		«EXPAND cTypeDefinition FOR ((PrimitiveDataType)this).type»
	«ENDIF»
«ENDDEFINE»
```

Şekil 60 typeDefinition şablonu.

Veri modelleri içinde tanımlanan birim tipleri MPI yordamlarında kullanırken MPI veri tipleri olarak kullanılması gerekmektedir. Bu yordamların dönüştürülmesinde kullanılmak üzere MPI veri tipi kod üretimi için mpiDataTypeDefinition şablonu oluşturulmuştur (Şekil 61).

```
«DEFINE mpiDataTypeDefinition FOR MpiDataType»
	«IF this.toString().endsWith("MpiByte")»MPI_BYTE«ENDIF»
	«IF this.toString().endsWith("MpiChar")»MPI_CHAR«ENDIF»
	«IF this.toString().endsWith("MpiShort")»MPI_SHORT«ENDIF»
	«IF this.toString().endsWith("MpiInt")»MPI_INT«ENDIF»
	«IF this.toString().endsWith("MpiLong")»MPI_LONG«ENDIF»
	«IF this.toString().endsWith("MpiUnsigned")»MPI_UNSIGNED«ENDIF»
	«IF this.toString().endsWith("MpiUnsignedChar")»MPI_UNSIGNED_CHAR
	«ENDIF»
	«IF this.toString().endsWith("MpiUnsignedShort")»MPI_UNSIGNED_SHORT
	«ENDIF»
	«IF this.toString().endsWith("MpiUnsignedLong")»MPI_UNSIGNED_LONG
	«ENDIF»
	«IF this.toString().endsWith("MpiFloat")»MPI_FLOAT«ENDIF»
	«IF this.toString().endsWith("MpiDouble")»MPI_DOUBLE«ENDIF»
	«IF
this.toString().endsWith("MpiLongDouble")»MPI_LONG_DOUBLE«ENDIF»
«ENDDEFINE»
```

Şekil 61 mpiDataTypeDefinition şablonu.

MPI yordamlarında MPI veri tiplerinin yanında türetilmiş veri yapılarının da kullanılması gerekmektedir. Bu sebeple MPI yordamında türetilmiş olması ya da MPI veri türü kullanılması durumuna göre dönüştürülmesi mpiTypeDefinition şablonu ile yapılmaktadır (Şekil 62).

```
«DEFINE mpiTypeDefinition FOR DataType»
	«IF this.metaType.name.endsWith("DerivedDataType")»
		«((DerivedDataType)this).name»
	«ELSE»
		«EXPAND mpiDataTypeDefinition FOR
		 ((PrimitiveDataType)this).type»
	«ENDIF»
«ENDDEFINE»
```

Şekil 62 mpiTypeDefinition şablonu.

Veri modellemesinde kullanılacak şablonlardan sonra türetilmiş verilerin kod içinde oluşturulabilmesi için bazı dönüşümlerin uygulanması gerekir. Türetilmiş veri tanımının yapılması için mpiDerivedTypeDefinition (Şekil 63), daha sonra da veri tipinin kod içinde türetilmesi için gereken MPI yordamının gerçekleştirilmesi için de mpiDerivedTypeOperation şablonu (Şekil 64) oluşturulmuştur.

```
«DEFINE mpiDerivedTypeDefinition FOR DataType»
	«IF this.metaType.name.endsWith("DerivedDataType")»
		MPI_Datatype «((DerivedDataType)this).name»;
	«ENDIF»
«ENDDEFINE»
```

Şekil 63 mpiDerivedTypeDefinition şablonu.

```
«DEFINE mpiDerivedTypeOperation FOR DerivedDataType»
      «IF this.type.toString().endsWith("Contiguous")»MPI_Type_contiguous
      «ELSEIF this.type.toString().endsWith("Vector")»MPI_Type_vector
      «ELSEIF this.type.toString().endsWith("Indexed")»MPI_Type_indexed
      «ELSEIF this.type.toString().endsWith("Struct")»MPI_Type_struct
      «ENDIF»
«ENDDEFINE»
```

Şekil 64 mpiDerivedTypeOperation şablonu.

MPI uygulaması içinde koşut çalışacak olan işlemleri oluşturan işlem maddeleri için kod dönüşümü yapılması gerekmektedir. Bu işlem maddeleri seri işlemler, durum kontrol işlemleri ve MPI yordamlarından oluşan koşut işleler olarak modellenmiştir. MPI toplu işlem yordamları bazı MPI toplu işlevlerini kullanmaktadır. Bu işlevlerin kod içinde gerçekleştirilmesi için mpiOperationDefinition şablonu oluşturulmuştur (Şekil 65).

```
«DEFINE mpiOperationDefinition FOR MpiOperation»
      «IF this.toString().endsWith("MpiMax")»MPI_MAX«ENDIF»
      «IF this.toString().endsWith("MpiMin")»MPI_MIN«ENDIF»
      «IF this.toString().endsWith("MpiSum")»MPI_SUM«ENDIF»
      «IF this.toString().endsWith("MpiProd")»MPI_PROD«ENDIF»
      «IF this.toString().endsWith("MpiLand")»MPI_LAND«ENDIF»
      «IF this.toString().endsWith("MpiBand")»MPI_BAND«ENDIF»
      «IF this.toString().endsWith("MpiLor")»MPI_LOR«ENDIF»
      «IF this.toString().endsWith("MpiBor")»MPI_BOR«ENDIF»
      «IF this.toString().endsWith("MpiLxor")»MPI_Lxor«ENDIF»
      «IF this.toString().endsWith("MpiBxor")»MPI_Bxor«ENDIF»
      «IF this.toString().endsWith("MpiMaxloc")»MPI_MAXLOC«ENDIF»
      «IF this.toString().endsWith("MpiMinloc")»MPI_MINLOC«ENDIF»
«ENDDEFINE»
```

Şekil 65 mpiOperationDefinition şablonu.

Koşut işlemler içinde tanımlı işlem maddeleri için tanımlanan actionDefinition şablonu, seri işlemler, durum kontrolleri ve MPI işlemleri için kod dönüşümlerini gerçekleştirmektedir. Şekil 66'da oluşturulan MPI modelden MPI kod dönüşümü şablonu verilmektedir.

```
«DEFINE actionDefinition FOR mpimetamodel::Action»
      «IF this.metaType.name.endsWith("SerialAction")»
                  «((SerialAction)this).name»;
      «ELSEIF this.metaType.name.endsWith("ForState")»
            for(forstateval=«((ForState)this).firstvalue»;
            forstateval «((ForState)this).state»;
            forstateval += «((ForState)this).ramp»)
            {
            «FOREACH ((StateAction)this).actions AS action»
                  «EXPAND actionDefinition FOR action»
            «ENDFOREACH»
            }
      «ELSEIF this.metaType.name.endsWith("WhileState")»
            while(«((WhileState)this).state»)
            {
            «FOREACH ((StateAction)this).actions AS action»
                  «EXPAND actionDefinition FOR action»
            «ENDFOREACH»
            }
      «ELSE»
            ... MPI Action Definitions
      «ENDIF»
«ENDDEFINE»
```

Şekil 66 MPI modelden MPI kod dönüşümü şablonu.

Koşut işlemde en sık kullanılan kütüphanelerden biri olan MPI için model-güdümlü yazılım geliştirme tekniklerinin uygulanması, bu kütüphane ile deneyimi az olan koşut işlem uygulaması geliştiricileri için yazılım geliştirme sürecini oldukça hızlandıracağı açıktır. MPI için oluşturulan üst model, bu kütüphane için alana özgü bir dil oluşturmaktadır.

4.2.2.6.3.2 NoC Durum Ayarlarının Üretilmesi

Oluşturulan NoC modeli ile her işlem birimine ait ayarların ne şekilde olduğu bulunmuştu. Bu aşamadan sonra yapılması gereken ise ilgili NoC için yapılandırma ayarlarının donanımlara uygun biçimde elde edilmesidir. Kullanılacak olan donanımlar genellikle bu tür ayarları metin tabanlı olarak kullanmaktadır. Bu sebeple aynı MPI kod üretimi gibi modelden metne dönüşüm ile elde edilebilir.

5. ARAÇ DESTEĞİ

Tez kapsamında tanımlanan koşut algoritmaların koşut hesaplama platformalarına atanması için model güdümlü yazılım geliştirme yaklaşımının gerçekleştirilmesi için Parmapper aracını hazırlanmıştır [48]. Java tabanlı olan bu araç için belirlenen kavramsal tasarım Şekil 67'de verilmiştir. Parmapper aracı iki farklı kesimden oluşmaktadır:

- Kütüphane Tanımlama Aracı
- Koşut Algoritma Atama Aracı

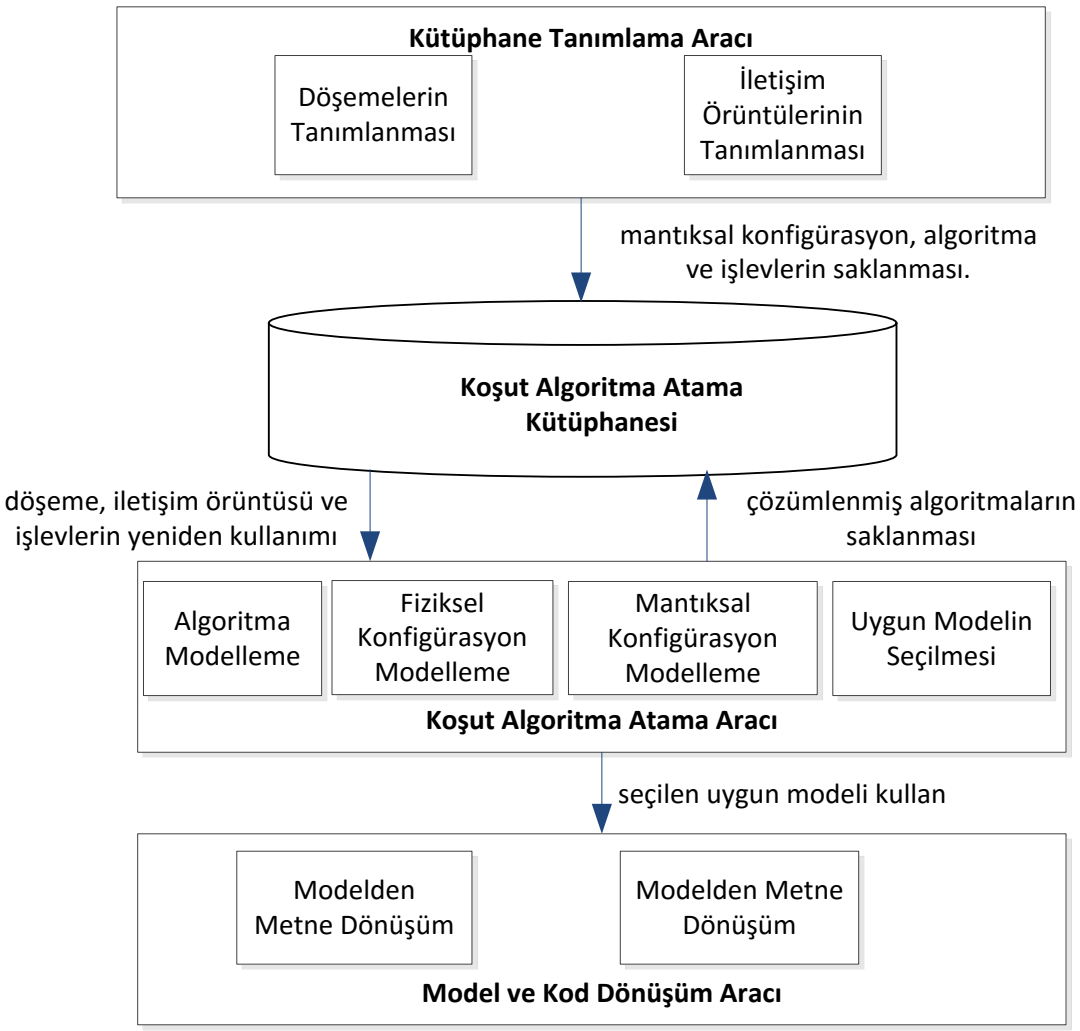
Bu iki kesimin dışındaki Model ve Kod Dönüşüm Aracı olarak ATL [47] ve XPand [26] gibi üçüncü parti araçlar kullanılmıştır.

Kütüphane Tanımlama Aracı yaklaşımda anlatılan döşemelerin ve iletişim örüntülerinin tanımlanması ve ilgili işlevler ile ilişkilendirilmesini sağlar. Bu tanımlamalar Koşut Algoritma Atama Kütüphanesine kaydedilir. Koşut Algoritma Atama Aracı ise algoritma ve fiziksel konfigürasyonlarının tanımlanmasını, ayrıca Koşut Algoritma Atama Kütüphanesi kullanılarak mantıksal konfigürasyonların oluşturulması ve seçilmesini sağlar. Seçilen uygun model ise Model ve Kod Dönüşüm Aracına girdi sağlanarak amaçlanan kodun üretilmesi sağlanır.

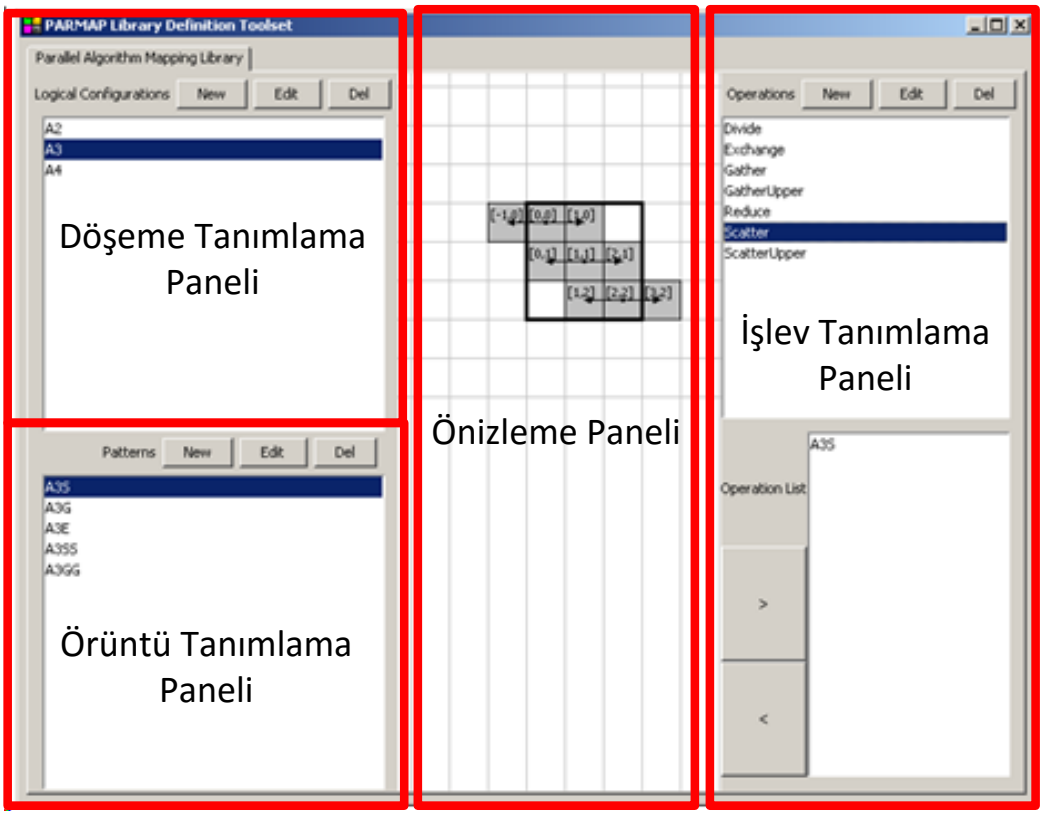
Şekil 68'de Parmapper Kütüphane Tanımlama Aracının ekran görüntüsü verilmiştir. Kütüphane Tanımlama Aracı dört ana panelden oluşmaktadır. İlk panel, Döşeme Tanımlama Panelidir. Burada tanımlı döşemeler listelenir. Yeni bir döşeme tanımlanmak istendiğinde ya da listeden bir döşeme güncellenmek istendiğinde döşeme oluşturma ekranı açılır. Burada istenen boyutta ve şekilde döşeme oluşturulur, hakim düğümleri belirlenir ve isim verilerek kaydedilir. İkinci panel olan Örüntü Tanımlama Panelinde ise Döşeme listesinde seçili olan döşemeye ait iletişim örüntüleri listelenir. Yeni bir örüntü tanımlanacağında ya da bir örüntü güncelleneceği zaman örüntü tanımlama ekranı açılır. Bu ekrandan düğümler arası iletişimler iletilecekleri yol ve yöne göre tanımlanır ve örüntüye isim verilerek kaydedilir. Üçüncü panel olan Önizleme Panelinde önceki bölümlerde bahsedilen kayıtlı döşeme ya da örüntülerden seçili olanın önizlemesi gösterilir. Son panel olan İşlev Tanımlama Panelinde ise yeniden kullanılabilir işlevlere isim verilirek kaydedilir ve örüntü panelinden seçilen örüntülerin bu işlevlere atanması sağlanır. Tüm bu

tanımlamalar Koşut algoritma Atama Kütüphanesine gerçek zamanlı olarak kaydedilir.

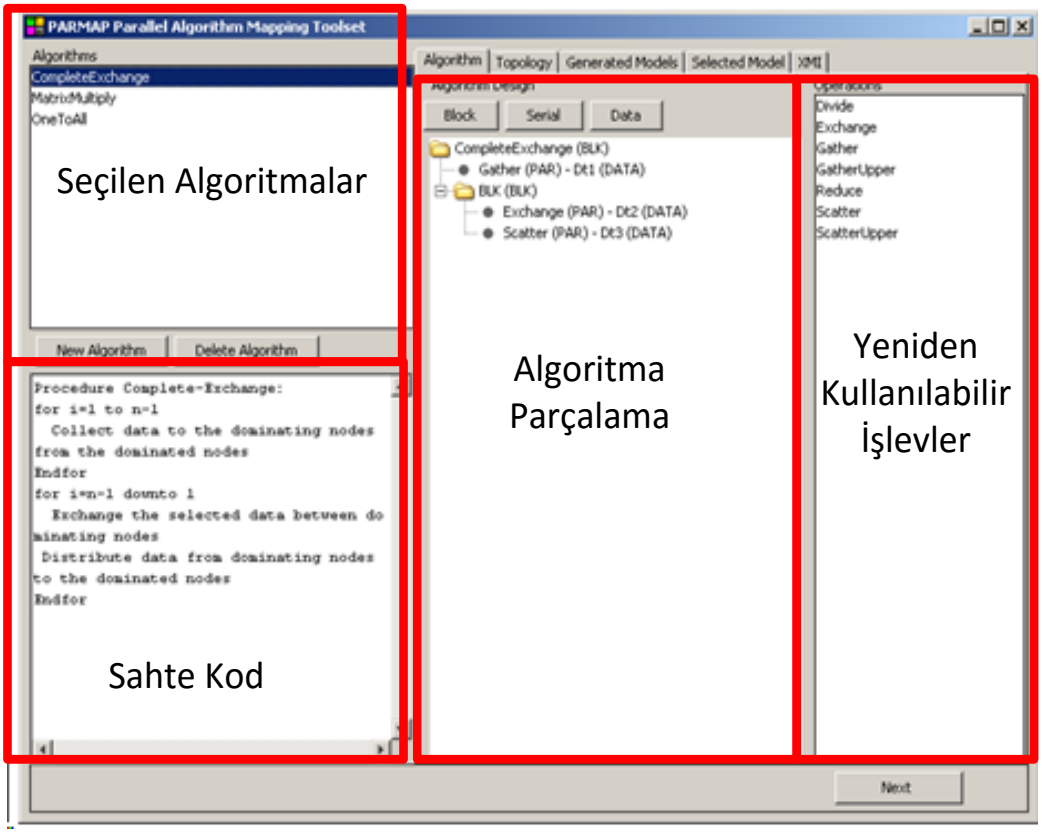
Şekil 69'da Parmapper Koşut Algoritma Atama Aracı ekran görüntüsü verilmiştir. Bu araç sıralı olarak yaklaşım adımlarını gerçekleştirecek panellerden oluşmaktadır. Şekil 69'da verilen ekran görüntüsünde algoritma parçalama paneline ait paneller gösterilmiştir. Tanımlı algoritmalar listede tutulur, sözde programları ile yaptıkları işlevler tanımlanır. Çalıştırma bölümlerinde koşut işlevlerden yeniden kullanılacak işlevler seçilir ve algorita çözümlemesi gerçekleştirilir. Fiziksel konfigürasyon panelinde konfigürasyon boyutları ve ölçekleme çarpanları belirlenir. Daha sonra mantıksal konfigürasyon seçenekleri oluşturulur. Bu seçenekler önizleme panelinde görsel olarak da sergilenir. Son olarak da koşut model seçilen mantıksal konfigürasyondan oluşturulur.



Şekil 67 Parmapper Aracı Kavramsal Tasarımı.



Şekil 68 Parmapper Kütüphane Tanımlama Aracı Ekran Görüntüsü.



Şekil 69 Parmapper Algoritma Atama Aracı Ekran Görüntüsü.

6. ÖRNEK ÇALIŞMALAR

6.1. VEKTÖREL TOPLAMA ALGORİTMASI

Vektörel işlemler koşut işlemin en çok kullanıldığı alanlardan birisidir. Vektörel toplama da iyi bilinen koşut algoritmalardan birisidir. Vektörel toplamada amaç toplama yapılacak vektörün koşut işlem yapacak işlem birimlerine dağıtılması, dağıtma sonrasında toplama işleminin gerçekleştirilmesi, sonrasında da sonuçların alınarak birleştirilmesinden oluşur. Vektörel toplama algoritmasının sıralı algoritması Şekil 70'de verilmiştir.

```
Procedure ArrayInc(A[], n):
if n=1 then
  *A += 1
else
  ArrayInc(A, n/2)
  ArrayInc(A+n/2, n)
Endif
```

Şekil 70 Vektörel toplama algoritması.

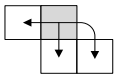
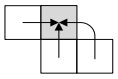
Algoritmanın çözümlenmesinde dört ayrı çalıştırma bölümü belirlenmiştir. Bu çalıştırma bölümlerinden oluşan vektörel toplama algoritması parçalama görünümü Çizelge 10'da verilmiştir. Algoritmanın ilk çalıştırma bölümü vektörü iki ayrı parçaya ayıran sıralı bir çalıştırma bölümüdür. Daha sonraki bölümde bu alt vektörler koşut olarak işlem birimlerine dağıtılır. Bu çalıştırma bölümü *Scatter* işlevini gerçekleştirir. Üçüncü çalıştırma bölümü toplama işlemini yapan sıralı bölümdür. Son bölüm ise sonuçları toplayan *Gather* işlevini gerçekleştiren koşut çalıştırma bölümüdür.

Çizelge 10 Vektörel toplama algoritması parçalama görünümü

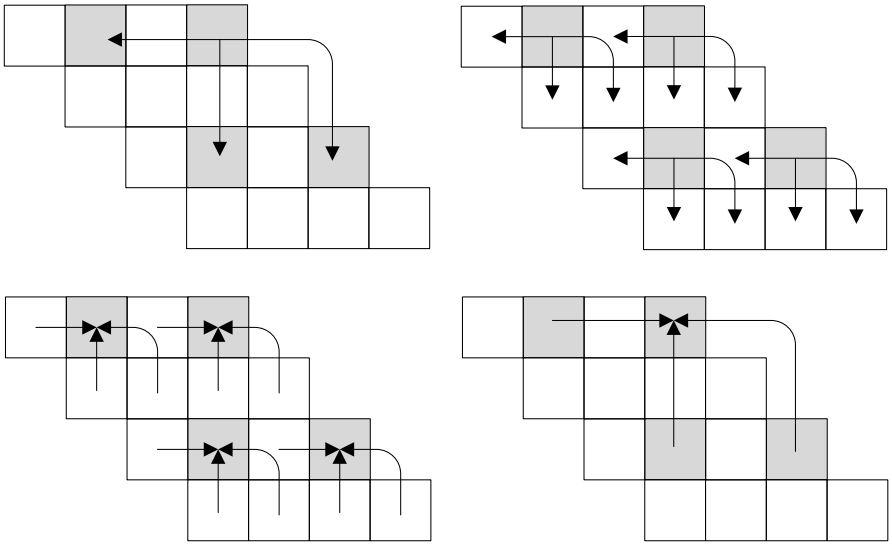
In.	Algorithm Section	Section Type	Operation
1	A1 = [A, A+n/2] A2 = [A+n/2, A+n]	SER	Decompose
2	Alt vektörleri dağıt	PAR	Scatter
3	*A += 1	SER	Increment
4	Vektörel sonuçları topla	PAR	Gather

Algoritmayı bu şekilde parçaladıktan sonra mantıksal konfigürasyona atamak için döşeme ve iletişim örüntülerini tanımladığımız planımızı oluşturmamız gerekir. Bu tanımları içeren vektörel toplama algoritmasından mantıksal konfigürasyona atama görünümü Çizelge 11'de verilmektedir. Çizelgede *Scatter* ve *Gather* işlevlerine ait iletişim örüntüleri 2x2 döşemeye göre planlanmıştır. Bu plana uygun olarak da ölçekleme startejileri de belirlenmiştir.

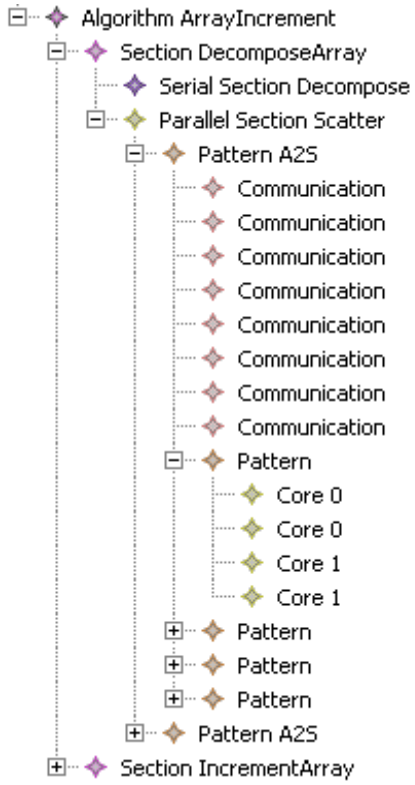
Çizelge 11 Vektörel toplama algoritmasından mantıksal konfigürasyona atama görünümü

In.	Algorithm Section	Plan	Scaling Strategy
1	A1=[A, A+n/2] A2=[A+n/2, A+n]	Her düğümde çalışır	N/A
2	Alt vektörleri dağıt		DOWN
3	*A += 1	Her düğümde çalışır	N/A
4	Vektörel sonuçları topla		UP

Plan yapıldıktan sonra mantıksal konfigürasyonlar vektör toplama algoritması için oluşturulur. Burada örnek 4x4 bir fiziksel konfigürasyona uygun olarak mantıksal konfigürasyon oluşturulmuştur. Sonuç olarak oluşturulan model Şekil 71'de gösterilmektedir. İlk iki mantıksal konfigürasyon *Scatter* işlevinin adımlarını, son iki mantıksal konfigürasyon ise *Gather* işlevinin adımlarını vermektedir. Buna bağlı oluşan koşut modelin görünümü Şekil 72'de gösterilmektedir.



Şekil 71 Vektörel toplama algoritması mantıksal konfigürasyon görünümü



Şekil 72 Vektörel toplama algoritması koşut modeli

6.2. MATRİS ÇARPMA ALGORİTMASI

Matris çarpma algoritması çarpma işlemlerinin yoğun olarak yapıldığı ve karmaşıklığı üst düzey olan algoritmalardandır. Bu sebeple koşut işlem kullanılması en çok tercih edilen algoritmaların başında gelmektedir. Alt matrislere parçalayarak matris çarpma algoritması Şekil 73'de gösterilmektedir. Bu algoritmaya göre iki matris dört ayrı parçaya ayrılmaktadır. Bu alt matrisler birbirler ile çarpımı olacak şekilde sekiz ayrı matris çarpma işlemi özyineli olarak yapılmaktadır. Çıkan sonuçlar da dört farklı matris toplama işlemi ile toplanmakta ve sonuç matrisi oluşturulmaktadır.

Algoritmanın çözümlenmesinde dört ayrı çalıştırma bölümü belirlenmiştir. Bu çalıştırma bölümlerinden oluşan matris çarpma algoritması parçalama görünümü Çizelge 12'de verilmiştir. İlk çalıştırma bölümü alt matrisleri dağıtma işlevini gerçekleştirir. Sonraki bölümde sıralı olarak matris çarpma işlemi yapılır. Üçüncü bölüm matris çarpma sonuçlarının toplanmasını sağlar ve son olarak dördüncü bölümde matris çarpım sonuçları sıralı olarak toplanarak sonuç matrisi elde edilir.

```
Procedure Matrix-Multiply(A, B, s):
if s=1 then
  C = A * B
endif
P0 = Matrix-Multiply(A00, B00, s-1)
P1 = Matrix-Multiply(A01, B10, s-1)
P2 = Matrix-Multiply(A00, B01, s-1)
P3 = Matrix-Multiply(A01, B11, s-1)
P4 = Matrix-Multiply(A10, B11, s-1)
P5 = Matrix-Multiply(A11, B10, s-1)
P6 = Matrix-Multiply(A10, B01, s-1)
P7 = Matrix-Multiply(A11, B11, s-1)
C00 = P0 + P1
C01 = P2 + P3
C10 = P4 + P5
C11 = P6 + P7
```

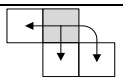
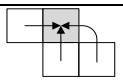
Şekil 73 Matris çarpma algoritması.

Çizelge 12 Matris çarpma algoritması parçalama görünümü

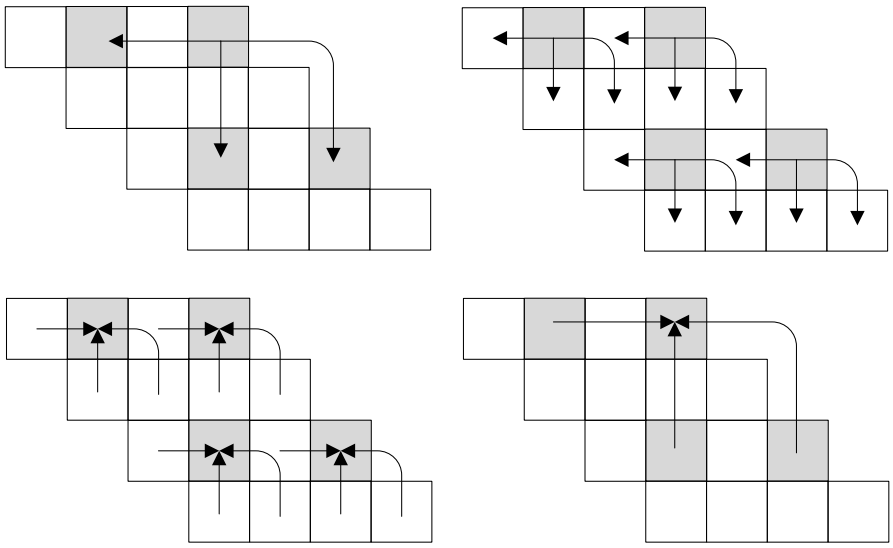
In.	Algorithm Section	Section Type	Operation
1	Alt matrisleri dağıt	PAR	Scatter
2	C = A * B	SER	Multiply
3	Matris çarpım sonuçlarını topla	PAR	Gather
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	SER	Sum

Algoritmayı bu şekilde parçaladıktan sonra mantıksal konfigürasyona atamak için döşeme ve iletişim örüntülerini tanımladığımız planımızı oluştururuz. Bu tanımları içeren matris çarpma algoritmasından mantıksal konfigürasyona atama görünümü Çizelge 13'de verilmektedir. Çizelgede *Scatter* ve *Gather* işlevlerine ait iletişim örüntüleri 2x2 döşemeye göre planlanmıştır. Bu plana uygun olarak da ölçekleme startejileri de belirlenmiştir.

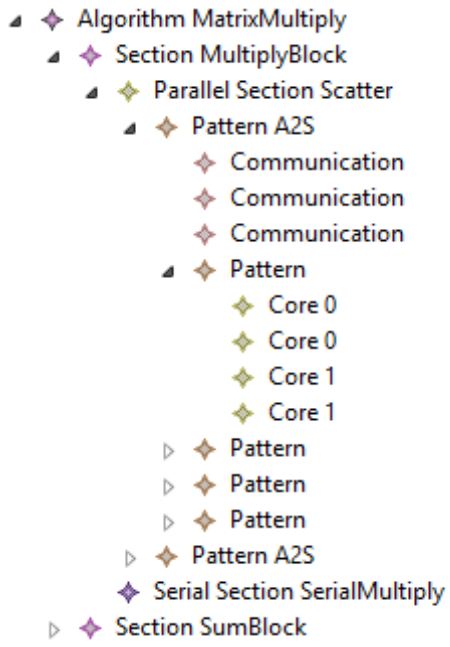
Çizelge 13 Matris çarpma algoritmasından mantıksal konfigürasyona atama görünümü

In.	Algorithm Section	Plan	Scaling Strategy
1	Alt matrisleri dağıt		DOWN
2	C = A * B	Her düğümde çalışır	N/A
3	Matris çarpım sonuçlarını topla		UP
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	Her düğümde çalışır	N/A

Plan yapıldıktan sonra mantıksal konfigürasyonlar vektör toplama algoritması için oluşturulur. Burada örnek 4x4 bir fiziksel konfigürasyona uygun olarak mantıksal konfigürasyon oluşturulmuştur. Sonuç olarak oluşturulan model Şekil 74'de gösterilmektedir. İlk iki mantıksal konfigürasyon *Scatter* işlevinin adımlarını, son iki mantıksal konfigürasyon ise *Gather* işlevinin adımlarını vermektedir. Buna bağlı oluşan koşut modelin görünümü Şekil 75'de gösterilmektedir.



Şekil 74 Matris çarpma algoritması mantıksal konfigürasyon görünümü



Şekil 75 Matris çarpma algoritması koşut modeli

6.3. MATRİS DEVRİĞİ ALGORİTMASI

Bir matrisin devriği (*transpose*) satırların sütun, sütunların satır haline getirilmesiyle elde edilen matristir. Matris devriği algoritması Şekil 76'da verilmektedir. Matris devriği bulunurken matrisin bir elemanı ilgili hücrenin karşılığındaki köşegene gönderilmesi sağlanır. Oradan da devrik hücresine iletilmesi sağlanır.

```
Procedure MatrixTranspose(A[]):
Do j to m-1
  k = j //Get diagonal position
  Do i to n-1
    Copy all blocks of A[i,j] to P(k, k)
    Send from P(k,k) to A[j,i]
  End do
End do
```

Şekil 76 Matris devriği algoritması.

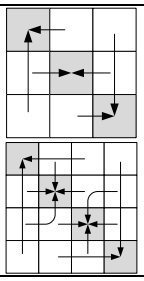
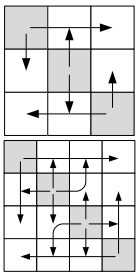
Matris devriği algoritması çözümlendiğinde verinin matris köşegeninde bulunan hakim düğüme gönderilmesi çalıştırma bölümü ve matris köşegeninden diğer düğüme gönderilmesi çalışma bölümü olmak üzere iki adet bölüme ayrılması gerekir. Buna göre tanımlanan matris devriği algoritması parçalama görünümü Çizelge 14'de verilmiştir. Bu iki çalıştırma bölümü de koşut olacaktır. Temel işlevleri de *Gather* ve *Scatter* işlevleri olarak tanımlanabilir.

Çizelge 14 Matris devriği algoritması parçalama görünümü

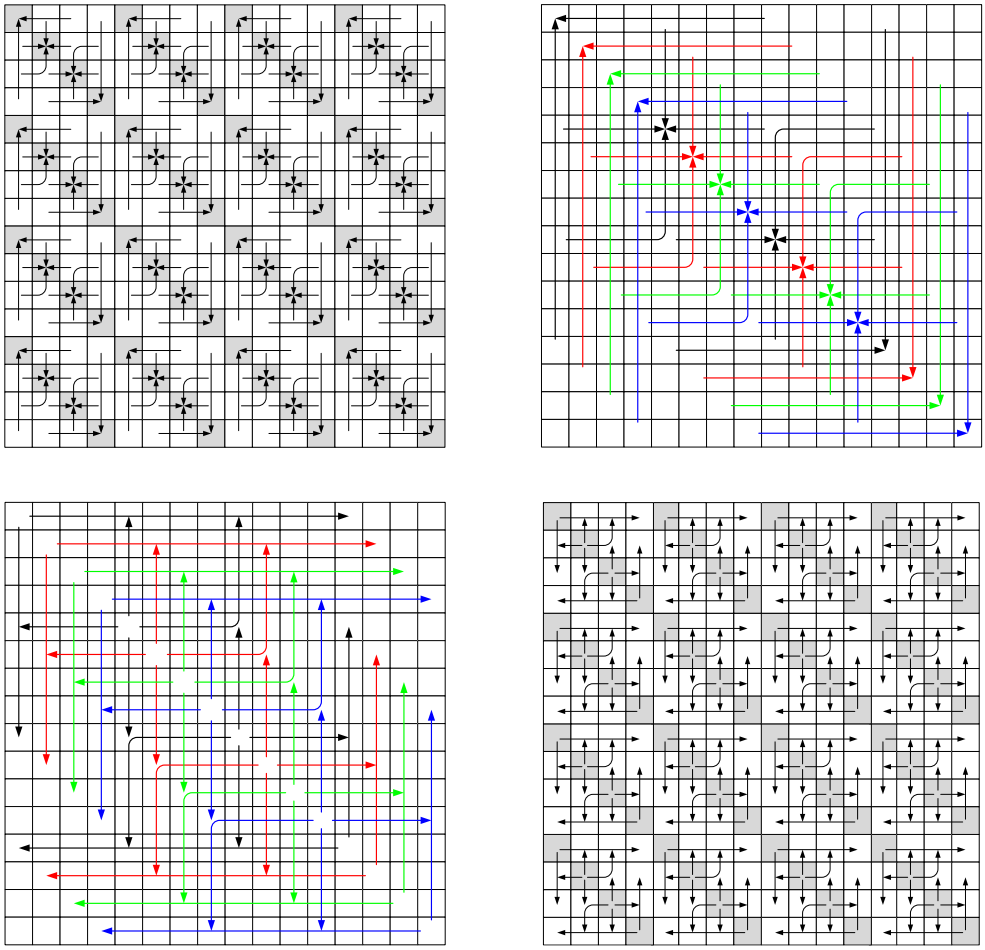
In.	Algorithm Section	Section Type	Operation
1	Copy all blocks of A[i,j] to P(k, k)	PAR	Gather
2	Send from P(k,k) to A[j,i]	PAR	Scatter

Algoritmayı bu şekilde parçaladıktan sonra mantıksal konfigürasyona atamak için döşeme ve iletişim örüntülerini tanımladığımız planımızı oluşturmamız gerekir. Bu tanımları içeren matris devriği algoritmasından mantıksal konfigürasyona atama görünümü Çizelge 15'de verilmektedir. Çizelgede *Scatter* ve *Gather* işlevlerine ait iletişim örüntüleri 3x3 ve 4x4 döşemelere göre planlanmıştır. Bu plana uygun olarak da ölçekleme startejileri de belirlenmiştir.

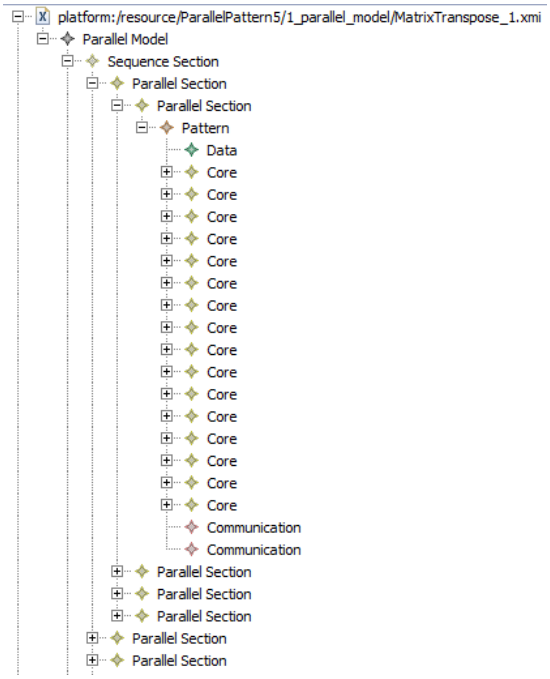
Çizelge 15 Matris devriği algoritmasından mantıksal konfigürasyona atama görünümü

In.	Algorithm Section	Plan	Scaling Strategy
1	Copy all blocks of A[i,j] to P(k, k)		UP
2	Send from P(k,k) to A[j,i]		DOWN

Plan yapıldıktan sonra mantıksal konfigürasyonlar vektör toplama algoritması için oluşturulur. Burada örnek 16x16 bir fiziksel konfigürasyona uygun olarak mantıksal konfigürasyon oluşturulmuştur. Sonuç olarak oluşturulan model Şekil 77'de gösterilmektedir. İlk iki mantıksal konfigürasyon *Gather* işlevinin adımlarını, son iki mantıksal konfigürasyon ise *Scatter* işlevinin adımlarını vermektedir. Buna bağlı oluşan koşut modelin görünümü Şekil 78'de gösterilmektedir.



Şekil 77 Matris devriği algoritması mantıksal konfigürasyon görünümü



Şekil 78 Matris devriği algoritması koşut modeli

6.4. TÜMÜNÜ EŞLEME ALGORİTMASI

Tümünü eşleme algoritması çoklu cisim (n-body) benzetimlerinde farklı cisimlerin kendi aralarındaki çekim kuvvetlerinin hesaplanması için birbirlerinin verilerini karşılıklı paylaştıkları bir algoritmadır. Özelikle moleküler benzetimler ya da uzaysal benzetimler gibi bilimsel hesaplamalarda çok fazla sayıda cizim için bu hesaplamalar yapıldığı için koşut işlemin kullanılması kaçınılmaz olmaktadır. Tümünü eşleme algoritması da bu cisimlere ait verilerin karşılıklı değişim yapmasını sağlayan algoritmadır.

```
Procedure AllPair(A[]):
For all nodes
  Send current position and forces to destination
  Get position and forces from destination
  Compute forces due to interactions
End for
```

Şekil 79 Tümünü eşleme algoritması.

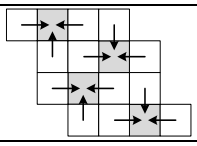
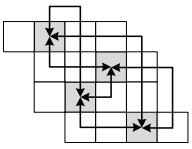
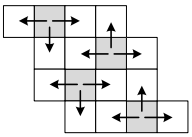
Tümünü eşleme algoritmasında üç farklı bölüm görünmektedir. Ancak verinin gönderilip alınmasında verilerin karşılıklı değiş tokuş yapılması için de bir bölüm ihtiyacı olmaktadır. Bu ek bölüm ile oluşturulan dört bölümlük tümünü eşleme algoritma parçalama görünümü Çizelge 16'da gösterilmektedir. Çizelgedeki ilk üç bölüm koşut bölümlerdir ve verinin karşılıklı eşlemesini sağlamaktadır. Son bölüm ise güçlerin hesaplanmasını sağlayan sıralı bölümdür.

Çizelge 16 Tümünü eşleme algoritması parçalama görünümü

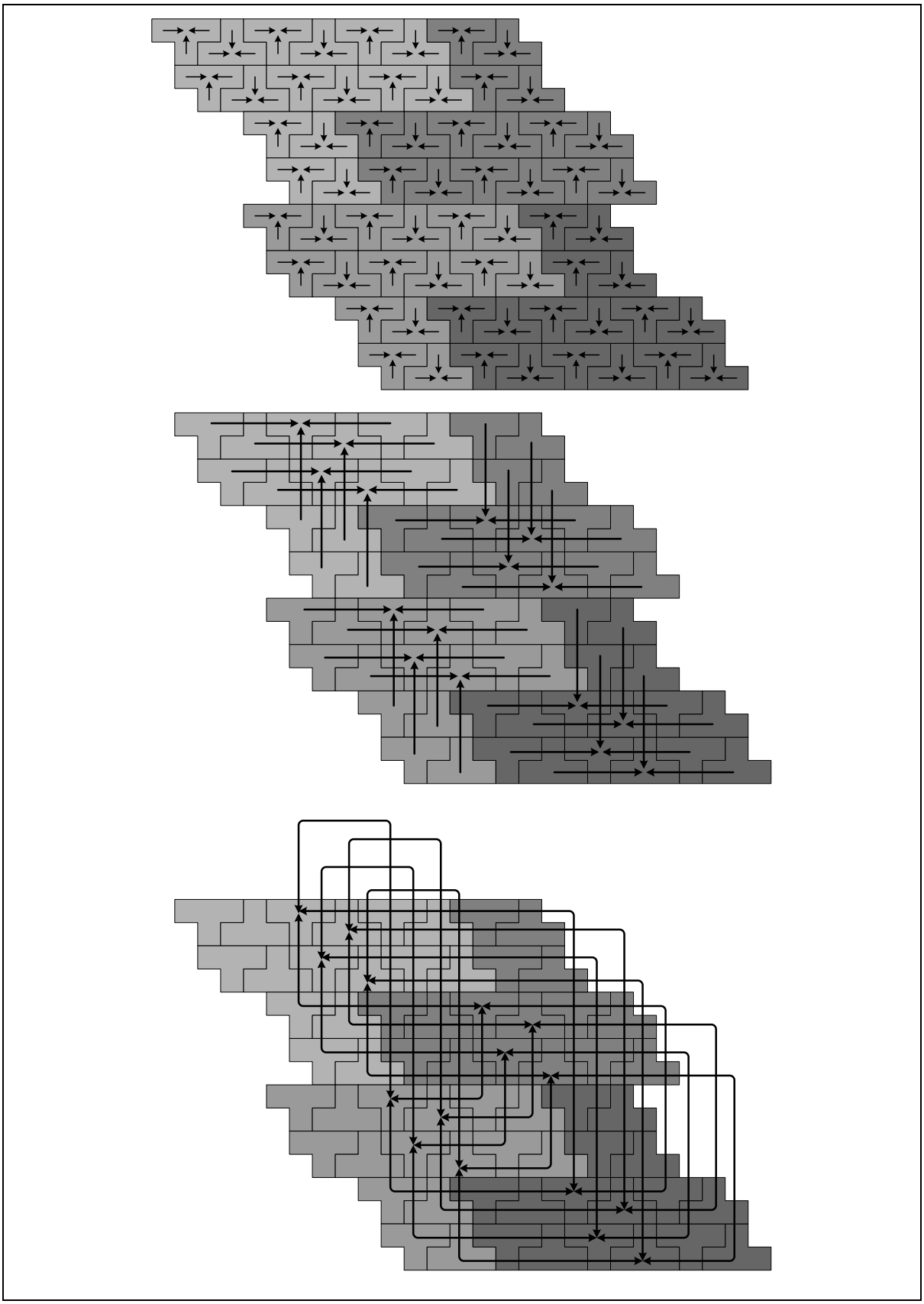
In.	Algorithm Section	Section Type	Operation
1	Veriyi hakim düğüme gönder	PAR	Scatter
2	Hakim düğümler arası veri değiştir	PAR	Exchange
3	Hakim düğümden veriyi al	PAR	Gather
4	Güçleri veriyi kullanarak hesapla	SER	Calculate

Algoritmayı bu şekilde parçaladıktan sonra mantıksal konfigürasyona atamak için döşeme ve iletişim örüntülerini tanımladığımız planımızı oluşturmamız gerekir. Bu tanımları içeren matris devriği algoritmasından mantıksal konfigürasyona atama görünümü Çizelge 15'de verilmektedir. Çizelgede *Scatter* ve *Gather* işlevlerine ait iletişim örüntüleri 3x3 ve 4x4 döşemelere göre planlanmıştır. Bu plana uygun olarak da ölçekleme startejileri de belirlenmiştir.

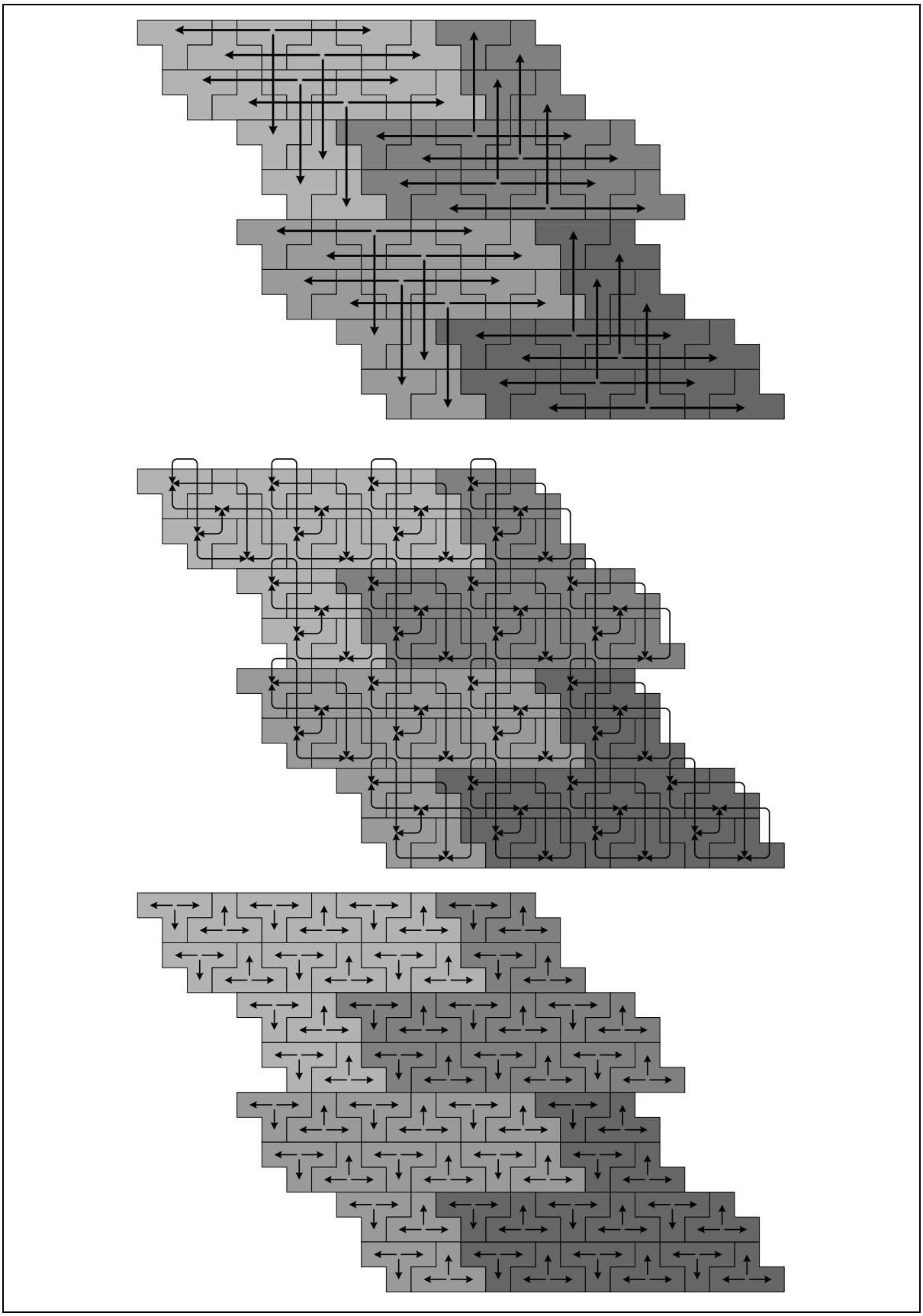
Çizelge 17 Matris devriği algoritmasından mantıksal konfigürasyona atama görünümü

In.	Algorithm Section	Plan	Scaling Strategy
1	Veriyi hakim düğüme gönder		DOWN
2	Hakim düğümler arası veri değiştir		UP
3	Hakim düğümden veriyi al		UP
4	Güçleri veriyi kullanarak hesapla	Her düğümde çalışacak	N/A

Plan yapıldıktan sonra mantıksal konfigürasyonlar vektör toplama algoritması için oluşturulur. Burada örnek 16x16 bir fiziksel konfigürasyona uygun olarak mantıksal konfigürasyon oluşturulmuştur. Tümünü eşleme için oluşturulan mantıksal konfigürasyon görünümü Şekil 80 ve Şekil 81'da gösterilmektedir.



Şekil 80 Tümünü eşleme algoritması mantıksal konfigürasyon görünümü (ilk üç adım).



Şekil 81 Tümünü eşleme algoritması mantıksal konfigürasyon görünümü (son üç adım).

7. DEĞERLENDİRME

Bir önceki kesimde koşut algoritmaların koşut hesaplama platformlarına atanması için tanımlanan model-güdümlü yazılım geliştirme yaklaşımı ile oluşturulmuş örnek çalışmalar verilmiştir. Bu kesimde bu çalışmalar ile ilgili değerlendirmelere yer verilmektedir.

Yapılan örnek çalışmalarla ilgili üç farklı durum değerlendirilmiştir. Öncelikle algoritmaların çözümlenmesi ve fiziksel konfigürasyonların tanımlanması sonucu oluşabilecek mantıksal konfigürasyon seçeneklerinin oluşturulma performansları ölçülmüştür. Daha sonra oluşturulan kaynak kodlar bir hesaplama platformunda denenmiştir, ve çalışılabilirliği değerlendirilmiştir. Son olarak da bir koşut hesaplama platformu benzetimi hazırlanmış ve kaynak kodların bu koşut hesaplama platformu benzetimindeki performansları ölçülmüş, modelden hesaplanan hızlanma ve etkinlik değerleri ile karşılaştırmalar yapılmıştır.

7.1. MANTIKSAL KONFİGÜRASYONLARIN OLUŞTURULMA PERFORMANSI

Değerlendirme aşamasında ilk olarak mantıksal konfigürasyonların oluşturulma performansına bakılmıştır. Bunun için hazırladığımız dört farklı algoritma için farklı boyutlarda fiziksel konfigürasyonlar için mantıksal konfigürasyonların oluşturulma süreleri ölçülmüştür. Bu ölçümler 8 çekirdekli Intel Xeon 2.67Ghz işlemci ve 32 GB RAM olan bir bilgisayar üzerinde yapılmıştır.

Çizelge 18 değerlendirilen dört farklı algoritmanın mantıksal konfigürasyon oluşturma performansı verilmektedir. Her algoritma için mantıksal konfigürasyondaki işlem birimi sayısı, oluşturulan mantıksal konfigürasyon sayısı ve oluşturma süreleri verilmektedir. Küçük boyutlu mantıksal konfigürasyonlar çok hızlı biçimde oluşturulduğu görülürken, en yüksek boyutlu ve en çok seçeneğin oluşturulduğu matris çarpma algoritmasında tüm mantıksal konfigürasyonlar yaklaşık iki buçuk saat gibi bir sürede oluşturulabilmektedir. Bu yaklaşımında bu boyutlarda uygun sürelerde üretim yapabildiğini göstermektedir. Daha güçlü bilgisayarlarda bu sürenin düşürülmesi mümkünken, bazı buluşsal yöntemler kullanarak da seçenek sayısı azaltılabilir.

Çizelge 18 Mantıksal konfigürasyonların oluşturulma performansı

Vektörel Toplama			Matris Çarpma		
İşlem Birimi Sayısı	#	Oluşturma Süresi (SS:dd:ss)	İşlem Birimi Sayısı	#	Oluşturma Süresi (SS:dd:ss)
144	3	00:00:00	144	3	00:00:00
1296	6	00:00:02	1296	6	00:00:02
5184	10	00:00:04	5184	10	00:00:04
46656	20	00:00:32	46656	20	00:00:36
186624	35	00:03:55	186624	35	00:04:17
419904	35	00:09:04	419904	35	00:10:18
746496	56	00:35:43	746496	56	00:39:04
Matris Devriği			Tümünü Eşleme		
İşlem Birimi Sayısı	#	Oluşturma Süresi (SS:dd:ss)	İşlem Birimi Sayısı	#	Oluşturma Süresi (SS:dd:ss)
64	8	00:00:01	144	2	00:00:00
256	16	00:00:03	1296	3	00:00:01
1024	32	00:00:08	5184	12	00:00:07
4096	64	00:00:24	46656	20	00:01:09
16384	128	00:01:58	186624	10	00:02:03
65536	256	00:15:23	419904	30	00:25:51
262144	512	02:21:33	746496	60	01:47:25

7.2. ÇOK ÇEKİRDEKLİ BİR PLATFORMDA ÇALIŞTIRMA

Sonraki aşamada, uygun mantıksal konfigürasyon seçeneklerinin oluşturulması ve bu modellerden kod dönüşümünün yapılması sonucu oluşturulan kaynak kodun bir hesaplama platformunda çalıştırılması değerlendirilmiştir. Bu çalışmalarda da kaynak kodun çalışma performansı ölçülmüş, hızlanma ve etkinlik değerleri hesaplanmıştır. Bu ölçümler 8 çekirdekli Intel Xeon 2.67Ghz işlemci ve 32 GB RAM olan bir bilgisayar üzerinde yapılmıştır. Yapılan ölçümler ve modelden hesaplanan metrikler Çizelge 19'de verilmiştir.

Burada çalışma performansı gerçek koşut hesaplama platformu ile aynı değildir. Ancak oluşturulan kodların doğru çalışabilirliğini görebilmek açısından iyi bir değerlendirme olmaktadır. Çizelgedeki değerlerin karşılaştırılmasında da doğru orantılı bir ölçüm olduğu görülmektedir. Ancak gerçek sonuçların elde edilmesi için gerçek bir koşut hesaplama platformunda ya da koşut hesaplama platformu benzetiminde çalıştırılması gerekmektedir.

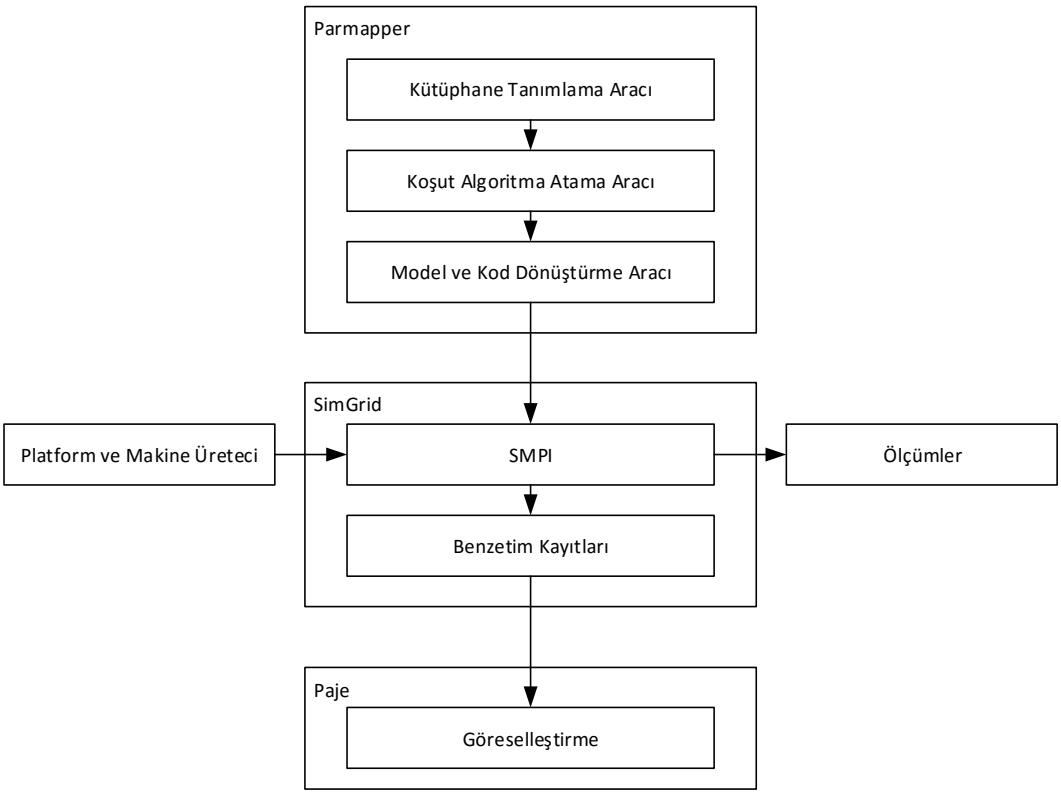
Çizelge 19 Algoritmaların çok çekirdekli platformda çalıştırılmasında hesaplanan metrikler ve ölçülen değerler

		Hesaplanan Metrikler					Ölçülen Değerler	
		PS	İU	İBS	Hesap.* Hızlan.	Hesap. Etk.*	Hızlan.*	Etk.*
Array Increment	1	316	38	200	6.3139t	3.16t	5.6050t	2.80t
	2	508	32	336	3.9321t	1.17t	5.2130t	1.55t
	3	556	20	370	3.5945t	0.97t	5.3895t	1.46t
Matrix Multiply	1	252	68	168	7.8939t	4.70t	7.1001t	4.23t
	2	252	64	168	7.8964t	4.70t	7.0824t	4.22t
	3	252	56	168	7.9014t	4.70t	7.1313t	4.24t
	4	252	52	168	7.9039t	4.70t	6.9186t	4.12t
	5	252	44	168	7.9089t	4.71t	7.1544t	4.26t
	6	252	40	168	7.9114t	4.71t	6.7183t	4.00t
	7	252	32	168	7.9164t	4.71t	6.8489t	4.08t
	8	252	28	168	7.9189t	4.71t	6.7760t	4.03t
Matrix Transpose	1	528	22	432	3.7847t	0.88t	5.1688t	1.20t
	2	528	22	480	3.7847t	0.79t	5.2792t	1.10t
	3	528	22	504	3.7847t	0.75t	5.2327t	1.04t
N-Body All Pair	1	792	42	516	2.5226t	0.49t	4.9980t	0.97t
	2	792	40	588	2.5227t	0.43t	4.9412t	0.84t
	3	792	36	624	2.5230t	0.40t	4.8724t	0.78t
	4	792	32	408	2.5232t	0.62t	4.9041t	1.20t
	5	792	36	444	2.5230t	0.57t	4.7648t	1.07t

Kısaltmalar: PS: Port Sayısı, İU: İletişim Uzunluğu, İBS: İşlem Birimi Sayısı
*t hesaplamanın tek bir işlemcide yapılan toplan süresi katsayısıdır.

7.3. BENZETİM PERFORMANSI

Oluşturulan kaynak kodların büyük ölçekli fiziksel konfigürasyonlarda denenmesi için SimGrid yüksek başarımlı hesaplama benzeticisi [49] kullanılmıştır. SimGrid dağıtık uygulamalar ve koşut işlemlerin türdeş olmayan ortamlarda benzetilmesi için bir uygulama çatısıdır. SimGrid SMPI adında MPI uygulamalarını tanımlı bir hesaplama platformunda çalıştırmak için benzetim altyapısı vardır. Şekil 82'de ürettiğimiz MPI kaynak kodlarını çalıştırmak için oluşturulan benzetim ortamı verilmiştir. Parmapper aracı ile üretilen MPI kaynak kodları SimGrid aracının SMPI bileşenine girdi olarak verilmektedir. Bunun yanında SMPI platform konfigürasyonu ve makine adı bilgilerine ihtiyaç duymaktadır. Bu amaçla platform ve makine konfigürasyon oluşturucu betikler hazırlanmıştır.



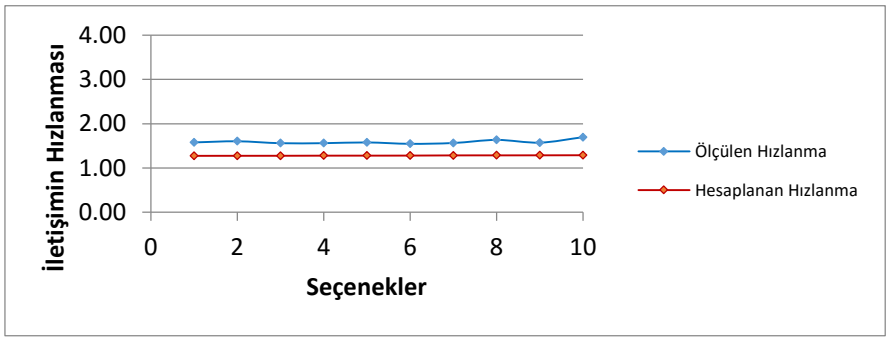
Şekil 82 Üretilen koşut MPI kodları için benzetim ortamı.

MPI kodlarının SMPI üzerinde çalıştırılması sonrasında benzetim izleme dosyaları ve ölçme sonuçları oluşturulmaktadır. Benzetim izleme dosyaları Paje [50] aracı kullanılarak görselleştirilebilmektedir. Paje çalışma anındaki tüm görevler için iletişim zamanlamalarını ve engelleme zamanlarını gösterebilmektedir. Şekil 83'de Paje kullanılarak oluşturulmuşi tümünü eşleme algoritması için görsel gösterim verilmektedir.

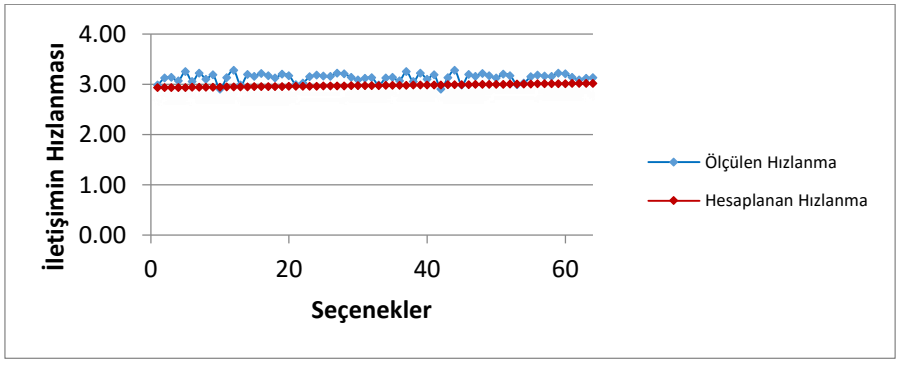
Şekil 83 Tümünü eşleme algoritması için görsel gösterim.

Bu benzetim yine 8 çekirdekli Intel Xeon 2.67Ghz işlemci ve 32 GB RAM olan bir bilgisayar üzerinde yapılmıştır. Bu benzetimin bilgisayar üzerindeki sınırlarından dolayı en büyük 5184 işlem birimi olan bir mantıksal konfigürasyon kaynak kodu

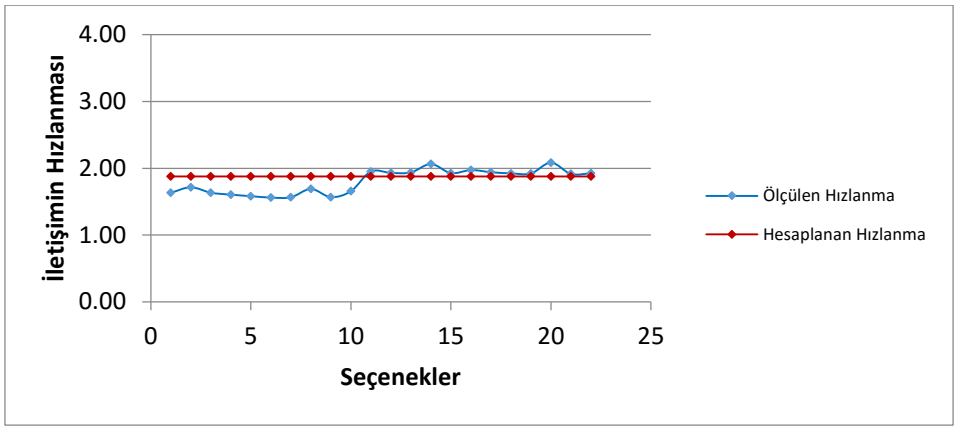
çalıştırılmıştır. Burada da çalışma süreleri ölçülerek gecikme süreleri belirlenmiştir. Bu süreler kullanarak boyut sıralı (*dimension-ordered*) dağıtma yöntemine göre iletişimin hızlanması değerleri hesaplanmıştır. SimGrid benzetiminde kullanılan sakla ve ilet kipine (*store-and-forward*) uygun olarak her iletişimde iletim maliyeti hesaplandıktan sonra boyut sıralı dağıtma gecikme süresinin algoritmanın toplam gecikme süresine oranı bulunmuştur. Bu değerler Şekil 84, Şekil 85, Şekil 86 ve Şekil 87'de dört farklı algoritma için benzetim performanslarına bağlı hesaplanan ve ölçülen iletişimin hızlanması değerleri olarak sunulmuştur. Şekillerde dikey değerler ölçülen ve hesaplanan iletişimin hızlanması değerlerini gösterirken, yatay değerler oluşturulan mantıksal konfigürasyon seçeneklerini göstermektedir. Grafiklerden benzetim sonuçlarının ve iletişimin hızlanması tahminlerinin modellerden teorik olarak hesaplanan değerler ile uyumlu olduğu da görülmektedir.



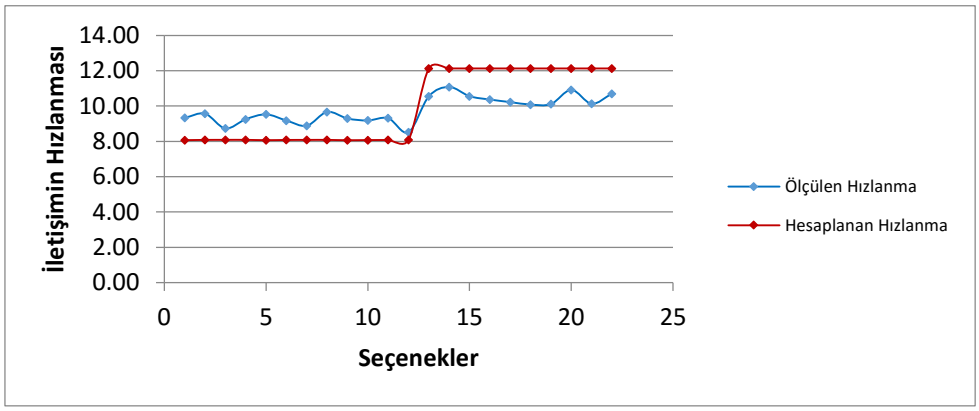
Şekil 84 Vektörel toplama algoritması için iletişimin hızlanması karşılaştırması.



Şekil 85 Matris çarpma algoritması için iletişimin hızlanması karşılaştırması.



Şekil 86 Matris devriği algoritması için iletişimin hızlanması karşılaştırması.



Şekil 87 Tümünü eşleme algoritması için iletişimin hızlanması karşılaştırması.

8. SONUÇ

Tez kapsamında koşut işlem için kullanılan koşut algoritmaların otomatik üretilmesi için model-güdümlü yazılım geliştirme yönteminin kullanımı anlatılmıştır. Koşut programlamada geleneksel yöntemlerin yerine model-güdümlü yaklaşımın kullanılması geliştirme sürecinde tasarımcıya kazanımlar sunmaktadır.

Koşut programlama dilleri ya da kütüphaneleri genellikle algoritmanın ya da verinin ne şekilde parçalanacağına yönelik yöntemler sunmaktadır. Ancak bu koşut programlama dillerinde çalışılacak topoloji ve bu topoloji için kullanılan ağ yapısı düşünülmemekte, algoritma parçalandıktan sonra adreslenen düğümlere nasıl dağıtılacağı belirlenmemektedir. Bu dağıtma sırasında oluşabilecek çakışmalar, kilitlenmeler gibi sorunlar görülememektedir ve bu da algoritmanın etkinliğini etkilemektedir. Tez kapsamında anlatılan modelleme yöntemi döşemeler ve bu döşemeler üzerindeki iletişim örüntülerinden oluşmakta ve küçük çaplı örüntülerde çakışma ve kilitlenme gibi problemler önceden görülebilmektedir. Yöntem aynı zamanda küçük döşemelerde tanımlanan örüntülerin özyineli ya da tekrarlı olarak ölçeklendiğinde bu problemlerin olmayacağını kanıtlamaktadır [29]. Bu sebeple geleneksel yöntemlere göre daha etkin iletişim ve algoritmaların oluşturulmasını sağlamaktadır.

Bu modelleme yönteminin sunduğu bir diğer avantaj da geleneksel programlama yöntemlerine göre tasarımcıya daha soyut bir gösterim sunabildiği için, algoritmaların farklı yöntemlerle de oluşturulmasına yardımcı olmaktadır.

Modelleme yöntemi belirlendikten sonra kullanılan gösterim standart bir gösterim haline gelmektedir. Bir topolojiye uygun olarak algoritma geliştirildiğinde boyutlarına bağlı olarak hangi döşemelerin ve iletişim örüntülerinin kullanılabileceği belirlenebilmektedir. Örneğin 15x15 düğümden oluşan bir örgüde 3x3 ve 5x5 döşeme ve iletişim örüntüleri kullanılabilmektedir. Bu gösterimin standart hale gelmesi, koşut işlem alanında kullanılabilecek bir alana özgü modelleme yapılmasını sağlamıştır. Bu modelleme yöntemi koşut işlem alanında genel bir sorun olan, bilgisayar bilimleri alanında deneyimi sınırlı olan farlı bilim alanlarından gelen koşut algoritma tasarımcılarının koşut algoritma oluşturma sıkıntısını gidermektedir.

Tasarımcı tarafından oluşturulan koşut algoritma modeli modelden modele dönüşüm ve modelden metne dönüşüm yöntemleri kullanılarak algoritma kaynak koduna dönüştürülebilmektedir. Bu noktada oluşturulan model önce hedef bir modele dönüştürülmektedir. Kullanılacak olan hedef koşut sistem üzerinde çalışabilen koşut kütüphaneye uygun olarak oluşturulan hedef model, tasarımcıya algoritma modelinin bu kütüphanelerden bağımsız olmasını sağlar. Aynı model farklı bir sistemde farklı bir kütüphane ya da dil için modelden modele dönüşüm ile kolaylıkla oluşturulabilir.

Hedef modele dönüştürüldükten sonra modelden metne dönüşüm yöntemleri ile kaynak kodun oluşturulması sağlanmaktadır. Bu kaynak kod otomatik üretim yöntemi ile oluşturulduğundan, elle yazılan bir kaynak koda göre hatalardan daha çok arınmış bir durumda olmaktadır.

Bu yöntem ile sadece koşut programlama dillerine uygun kaynak kod üretilmesi yapılmamaktadır. Aynı yöntem ile modellenen algoritma, iletişim altyapısı ve topoloji ile ilgili bilgileri de içerdiğinden, koşut sistemlerde kullanılan donanımlar ile ilgili dönüşümler de uygulanabilmektedir. Örnek olarak verilen NoC yongalarının anahtarlama durumları algoritmaya bağlı olarak belirlenebilmektedir. Bu sayede daha çok güç tüketen ve anahtarlamayı gelen veriye göre yaptığı için iletişim etkinliğini düşüren geleneksel anahtarlayıcılar yerine, NoC yongaları kullanılabilmekte ve algoritma özelinde zamana bağlı olarak durumları programlanabilmektedir. Bir diğer kazanç da algoritmada kullanılan iletişim örüntüsüne göre kullanılmayan yongaların o zaman zarfında güçlerinin düşürülerek ya da kesilerek güç tüketiminin azaltılması sağlanabilmektedir.

NoC yongalarının yerini de günümüzde daha hızlı oldukları için fotonic anahtarlayıcılardır. Bu anahtarlayıcılar ışığın içinde gönderilen veriyi istenilen istikamete yönlendirirler. Ancak verinin hangi yöne anahtarlanacağını gönderilen bilgiden alabilmek için fotonic bilginin elektronik bilgiye dönüştürülmesi gerekmektedir. Bunu veriye göre yapmak maliyetli olacağından daha üst düzeyden anahtarlayıcıların yönetilmesi gerekir. Tez kapsamında modellenen algoritmalardaki iletişim örüntülerinde yönlendirilecek olan istikamet önceden belirlenebildiği için verinin anahtarlanması için de bir model oluşturulmuş olur. Tezde sunulan yaklaşım bu anlamda fotonik anahtarlayıcılar için kullanılması mümkündür.

Tez kapsamında yapılan bu çalışma ile gerçekleştirilen kazanımlar özetlenecek olursa;

1. Koşut algoritmaların parçalanması ve adreslenmesi için daha etkin bir yöntem sunulmaktadır,
2. Koşut algoritmanın çalışması sırasında iletişim çakışmaları ya da kilitlenmeler gibi problemler önceden tespit edilebilmektedir,
3. Koşut algoritmanın oluşturulabilmesi için standart bir gösterim sunulmaktadır,
4. Standart gösterim ile oluşturulan modelleme sayesinde bilgisayar bilimleri alanı dışındaki bilimlerden insanların da kolaylıkla koşut algoritmalar oluşturulması sağlanabilmektedir,
5. Model-güdümlü yazılım geliştirme ile farklı koşut sistemler için farklı koşut programlama dillerinde ya da kütüphanelerinde otomatik kaynak kod üretimi gerçekleştirilmektedir,
6. Otomatik kaynak kod üretimi sayesinde hatasız koşut algoritmalar üretilebilmektedir,
7. Koşut kaynak kodlarının yanında, koşut sistemlerde kullanılan iletişime özgü donanımlara ait ayarların devingen yapıda oluşturulması sağlanarak daha etkin iletişim altyapısı sunulmaktadır,
8. Kullanılmayan anahtarlayıcıların güç kaynakları kapatılarak ya da azaltılarak koşut sistemin güç tüketimine yönelik iyileştirmeler yapılabilir,
9. Fotonik anahtarlayıcı gibi özel donanımlar için modelin sağladığı bilgiler kullanılarak eniyileştirmeler sağlanabilmektedir.

KAYNAKLAR

[1] Frank, M. P., The physical limits of computing. *Computing in Science & Engineering*, *4*(3), 16-26, **2002**.

[2] Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R.S., Yelick, K., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Keckler, S., Klein, D., Williams, R.S., Yelick, K., *Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. DARPA*, **2008**.

[3] Damian-Iordache, M., Pemmaraju, S.V., Automatic data decomposition for message-passing machines. In *Languages and Compilers for Parallel Computing* (pp. 64-78). Springer Berlin Heidelberg, **1998**.

[4] Peters, J.G., Spencer, C.C., S, V., Global communication on circuit-switched toroidal meshes. *Parallel Processing Letters*, *8*(02), 161-175, **1997**.

[5] Hirano, S., Kitsuregawa, M., Takagi, M., A high performance parallel I/O model and its deadlock prevention/avoidance technique on the Super Database Computer (SDC). In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on* (Vol. 1, pp. 21-30). IEEE, **1993**.

[6] Perego, R., Petris, G.D., Minimizing network contention for mapping tasks onto massively parallel computers. In *Parallel and Distributed Processing, 1995. Proceedings. Euromicro Workshop on* (pp. 210-218). IEEE, **1995**.

[7] Marowka, A., Analytic comparison of two advanced c language-based parallel programming models. In *Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004. Third International Workshop on* (pp. 284-291). IEEE, **2004**.

[8] Talia, D., Models and trends in parallel programming. *Parallel Algorithms and Application*, *16*(2), 145-180, **2001**.

[9] Kang, Z., Wanli, M., Graphical assistance in parallel program development. In *In: Proceedings of the 10th IEEE International Symposium on Visual*, **1994**.

[10] Moore, G.E., Cramming more components onto integrated circuits.*Proceedings of the IEEE*, *86*(1), 82-85, **1998**.

[11] Petersen, W.P., Arbenz, P., *Introduction to parallel computing* (No. 9). Oxford University Press, **2004**.

[12] Liao, X., Xiao, L., Yang, C., & Lu, Y. MilkyWay-2 supercomputer: system and application. Frontiers of Computer Science, 8(3), 345-356, **2014**.

[13] Gebali, F., *Algorithms and parallel computing* (Vol. 84). John Wiley & Sons, **2011**.

[14] Flynn, M.J., Very high-speed computing systems. *Proceedings of the IEEE*, *54*(12), 1901-1909, **1966**.

[15] Grama, A., Gupta, A., Karypis, G., *Introduction to parallel computing: design and analysis of algorithms*. Redwood City, CA: Benjamin/Cummings Publishing Company, **1994**.

[16] Sarkar, V., Amarasinghe, S., Campbell, D., Carlson, W., Chien, A., Dally, W., Elnohazy, E., Hall, M., Harrison, R., Harrod, W., Hill, K., Hiller, J., Karp, S., Koelbel, C., Koester, D., Kogge, P., Levesque, J., Reed, D., Schreiber, R., Richards, M., Scarpelli, A., Shalf, J., Snavely, A., Sterling, T., *Exascale Software Study: Software Challenges in Extreme Scale Systems. DARPA*, **2009**.

[17] Brown, A., Conallen, J., Tropeano, D., Introduction: Models, modeling, and model-driven architecture (mda). In *Model-Driven Software Development* (pp. 1-16). Springer Berlin Heidelberg, **2005**.

[18] Czarnecki, K., Antkiewicz, M., Kim, C.H.P., Lau, S., Pietroszek, K., Model-driven software product lines. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 126-127). ACM, **2005**.

[19] Czarnecki, K., Helsen, S., Feature-based survey of model transformation approaches. *IBM Systems Journal*, *45*(3), 621-645, **2006**.

[20] OMG, Model Driven Architecture (MDA), ormsc/2001-07-01, **2001**.

[21] Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., Neema, S., Developing applications using model-driven design environments. *Computer*, *39*(2), 33-40, **2006**.

[22] Booch, G., Jacobson, I., Rumbaugh, J., eds., *Omg Unified Modelling Language Specification V1.3*, **2000**.

[23] Gronback, R., Merks, E., Eclipse Modelling Project Webinar http://www.eclipse.org/community/training/webinars/080327_Modeling_Webinar.pdf, **2008**.

[24] Eclipse, Eclipse Modeling Framework (EMF) Core, http://www.eclipse.org/modeling/emf/, **2015**.

[25] OMG, Meta Object Facility (MOF) Core Specification, formal/2011-08-07, **2011**.

[26] Eclipse, Xpand, http://wiki.eclipse.org/Xpand, **2015**.

[27] Gelernter, D., Carriero, N., Coordination languages and their significance. *Communications of the ACM*, *35*(2), 96, **1992**.

[28] Hasselbring, W., Programming languages and systems for prototyping concurrent applications. *ACM Computing Surveys (CSUR)*, *32*(1), 43-79, **2000**.

[29] İmre, K., Baransel, C., Artuner, H., Efficient and scalable routing algorithms for collective communication operations on 2D all-port torus networks. *International Journal of Parallel Programming*, *39*(6), 746-782, **2011**.

[30] Baransel, C., İmre, K., A Parallel Implementation of Strassen's Matrix Multiplication Algorithm for Wormhole-Routed All-Port 2d Torus Networks. *The Journal of Supercomputing*, *62*(1), 486-509, **2012**.

[31] Bézivin, J., Heckel, R., Guest editorial to the special issue on language engineering for model-driven software development. *Software and Systems Modeling*, *5*(3), 231-232, **2006**.

[32] Guerra, E., de Lara, J., Malizia, A., Diaz, P., upporting user-oriented analysis for multi-view domain-specific visual languages. *Information and Software Technology*, *51*(4), 769-784, **2009**.

[33] Kühne, T., Matters of (meta-) modeling. *Software & Systems Modeling*,*5*(4), 369-385, **2006**.

[34] Meservy, T.O., Fenstermacher, K.D., Transforming software development: an MDA road map. *Computer*, (9), 52-58, **2005**.

[35] García-Magariño, I., Fuentes-Fernández, R., Gómez-Sanz, J.J., A framework for the definition of metamodels for Computer-Aided Software Engineering tools. *Information and Software Technology*, *52*(4), 422-435, **2010**.

[36] Kowalski, M., Wilkosz, K., A Domain Specific Language in Dependability Analysis. In *Dependability of Computer Systems, 2009. DepCos-RELCOMEX'09. Fourth International Conference on* (pp. 324-331). IEEE, **2009**.

[37] Amatriain, X., Arumi, P., Frameworks generate domain-specific languages: A case study in the multimedia domain. *Software Engineering, IEEE Transactions on, 37(4), 544-558*, **2011**.

[38] White, S. A., *BPMN modeling and reference guide: understanding and using BPMN*. Future Strategies Inc., **2008**.

[39] ISO/IEC 42010, *Systems and software engineering - Recommended practice for architectural description of software-intensive systems*, **2007**.

[40] Tekinerdogan, B., Arkin, E., Architecture Framework for Mapping Parallel Algorithms to Parallel Computing Platforms. In *MDHPCL@ MoDELS* (pp. 63-62), **2013**.

[41] Eclipse, Eclipse IDE, http://www.eclipse.org/, **2015**.

[42] Lenstra, H. W., & Pomerance, C., A rigorous time bound for factoring integers. *Journal of the American Mathematical Society*, *5*(3), 483-516, **1992**.

[43] Hoffmann, A., & Neubauer, B., Deployment and configuration of distributed systems. In *System Analysis and Modeling* (pp. 1-16). Springer Berlin Heidelberg, **2005**.

[44] Stawinska, M., Kurzyniec, D., Stawinski, J., & Sunderam, V., Automated Deployment Support for Parallel Distributed Computing. In *Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on* (pp. 139-146). IEEE, **2007**.

[45] Barney, B., Message Passing Interface (MPI) Tutorial https://computing.llnl.gov/tutorials/mpi/, **2012**.

[46] Sgroi, M., Sheets, M., Mihal, A., Keutzer, K., Malik, S., Rabaey, J., Sangiovanni-Vencentelli, A., Addressing the system-on-a-chip interconnect woes through communication-based design. In *Proceedings of the 38th annual Design Automation Conference* (pp. 667-672). ACM, **2001**.

[47] Atlas, Atlas Transformation Language (Atl) http://www.eclipse.org/atl/, **2012**.

[48] Arkın, E., Tekinerdogan, B., İmre, K. M., Model-Driven Approach for Supporting the Mapping of Parallel Algorithms to Parallel Computing Platforms. In *Model-Driven Engineering Languages and Systems* (pp. 757-773). Springer Berlin Heidelberg, **2013**.

[49] Casanova, H., Giersch, A., Legrand, A., Quinson, M., & Suter, F., Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, *74*(10), 2899-2917, **2014**.

[50] de Kergommeaux, J. C., Stein, B., Bernard, P. E., Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, *26*(10), 1253-1274, **2000**.

ÖZGEÇMİŞ

Kimlik Bilgileri

Adı Soyadı : Ethem Arkın

Doğum Yeri : Berlin

Medeni Hali : Evli

E-posta :earkin@gmail.com

Adresi :ASELSAN A.Ş. SST Sektör Başkanlığı Macunköy / ANKARA

Eğitim

Lise :Ankara Cumhuriyet Lisesi (1997)

Lisans :Hacettepe Üniversitesi Bilgisayar Mühendisliği (2001)

Yüksek Lisans :Hacettepe Üniversitesi Bilgisayar Mühendisliği (2004)

Yabancı Dil ve Düzeyi

İngilizce İyi

İş Deneyimi

2005-Halen ASELSAN A.Ş.
Yazılım Mühendisi

2001-2005 Hacettepe Üniversitesi Bilgisayar Mühendisliği Bölümü
Araştırma Görevlisi

Deneyim Alanları

Yazılım Mimarileri, Model Güdümlü Yazılım Geliştirme, Yazılım Ürün Hattı, Koşut Programlama, Arakatman Yazılımları

Tezden Üretilmiş Projeler ve Bütçesi

Yoktur.

Tezden Üretilmiş Yayınlar

- Arkın, E., Tekinerdogan, B. "Parallel Application Development using Architecture View Driven Model Transformations", in MODELSWARD 2015 - 3rd International Conference on Model-Driven Engineering and Software

Development, Springer Communications in Computer and Information Science, to be published **2015**.

- Arkın, E., Tekinerdogan, B. "Architectural View Driven Model Transformations for Supporting the Lifecycle of Parallel Applications", in Proc. Of 3rd Int. Conf. On Model-Driven Engineering and Software Development (MODELSWARD '15), ISBN 978-989-758-083-3, pages 40-49, **2015**.
- Tekinerdogan, B., Arkın, E. "Architecture Framework for Modeling the Deployment of Parallel Applications on Parallel Computing Platforms", in Proc. Of 3rd Int. Conf. On Model-Driven Engineering and Software Development (MODELSWARD '15), ISBN 978-989-758-083-3, pages 185-192, **2015**.
- Arkın, E., Tekinerdogan, B. "Domain Specific Language for Deployment of Parallel Applications on Parallel Computing Platforms", in Proc. Of 2014 European Conference on Software Architecture Workshops (ECSAW '14), ACM New York, NY, USA, Article 16, **2014**.
- Arkın, E., Tekinerdogan, B., İmre, K. "Model-Driven Approach for Supporting the Mapping of Parallel Algorithms to Parallel Computing Platforms", in Proc. of ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems, Miami, Florida, US, Springer LNCS 8107, pp. 757–773, **2013**.
- Tekinerdogan, B., Arkın, E. "Architecture Framework for Mapping Parallel Algorithms to Parallel Computing Platforms", MDHPCL@MoDELS 2013, 53-62, **2013**.
- Arkın, E., Tekinerdogan, B. "Model-Driven Transformations for Mapping Parallel Algorithms on Parallel Computing Platforms", MDHPCL@MoDELS 2013, 63-72, **2013**.

Tezden Üretilmiş Tebliğ ve/veya Poster Sunumu ile Katıldığı Toplantılar

- 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015)
- 2014 European Conference on Software Architecture Workshops (ECSAW 2014)

- 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)
- 3rd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing (MDHPCL 2013)