

Komputasi Paralel

Pekan 3
Paralel Programming

Definisi

- Task & Parallel Task
- Serial Execution & Parallel Execution
- Shared Memory & Distributed Memory
- Communication & Locality
- Synchronization
- Granularity
- Observed Speedup
- Parallel Overhead

Task

- **Task** : bagian dari pekerjaan komputasi yang diskrit. bisa berupa program atau instruksi yang dieksekusi oleh prosesor
- **Parallel Task** : task yang dapat dikerjakan oleh beberapa prosesor dan tetap menghasilkan hasil yang benar.

Execution

- **Serial Execution** : Eksekusi program secara sekuensial, sebuah statement pada satu waktu. semua program paralel memiliki bagian program yang harus dijalankan secara serial.
- **Parallel Execution** : Eksekusi program oleh lebih dari satu task. setiap task bisa mengeksekusi statement yang sama ataupun berbeda pada satu waktu.

Shared vs Distributed

- Sebagai arsitektur hardware
 - shared : semua prosesor memiliki akses fisik ke memori bersama
 - distributed : masing masing prosesor memiliki memori sendiri
- Sebagai model pemrograman
 - shared : setiap prosesor melihat memori bersama secara logicalnya, meski secara fisiknya memori tersebut terpisah
 - distributed : setiap task hanya melihat lokal memori dan harus berkomunikasi untuk mengakses memori pada prosesor lain.

Communication

- Cara task pada pemrograman paralel untuk bertukar data
 - message passing
 - shared memory
- Akses ke **memori lokal** (node yang sama) lebih murah dibandingkan akses ke **memori remote** (node yang berbeda).
- Sehingga akses ke data lokal seharusnya lebih sering dibandingkan data remote (***locality***)

Synchronization

- Koordinasi task paralel secara real time. Biasanya diimplementasikan dengan cara membuat titik sinkronisasi (synchronization poin) dimana task tidak bisa melanjutkan dulu sebelum semua task lain mencapai titik tersebut.
- Melibatkan lebih dari satu task, dan menyebabkan waktu proses program paralel meningkat

Granularity

- Ukuran kualitatif dari rasio komputasi terhadap komunikasi
 - **Coarse grained** : komputasi besar diantara komunikasi antar task
 - **Fine grained** : komputasi kecil diantara komunikasi antar task

Observed Speedup

- Indikator paling sederhana untuk melihat peningkatan performa program paralel

$$\frac{\text{waktu eksekusi program serial}}{\text{waktu eksekusi program paralel}}$$

Amdahl Law

- Tahun 1967, Gene Amdahl memformulasikan performance benefit yang bisa diperoleh dari sebuah program paralel. Dengan **mengasumsikan** sbb :
 - Ada aplikasi sekuensial yang perlu waktu sebanyak T untuk diproses pada CPU tunggal
 - Aplikasinya memiliki α bagian yang bisa diparalelkan ($0 \leq \alpha \leq 1$). sisanya tetap harus dilakukan secara sekuensial
 - Eksekusi program secara paralel tidak membutuhkan overhead komunikasi, dan bagian paralelnya dapat dibagi secara rata terhadap berapapun jumlah CPU.

Amdahl Law

- Dengan asumsi tersebut, speedup yang diperoleh dengan menggunakan N buah node adalah sebagai berikut

$$speedup = \frac{t_{seq}}{t_{par}} = \frac{T}{(1 - \alpha)T + \frac{\alpha \cdot T}{N}} = \frac{1}{1 - \alpha + \frac{\alpha}{N}}$$

- Batas speedup yang mungkin diraih adalah

$$\lim_{N \rightarrow \infty} (speedup) = \frac{1}{1 - \alpha}$$

Contoh

- Sebuah program dijalankan pada komputer CPU tunggal menghabiskan waktu 10 menit. Setelah dianalisa, ternyata 60% bagian program tersebut dapat diparalelkan.
- Berapa waktu yang dibutuhkan untuk menjalankan program tersebut pada komputer dengan 4 buah CPU?
- Berapa speedup yang diperoleh dan speedup maksimalnya?

Contoh

- $T = 10$ menit
- $\alpha = 60\% = 0.6$
- Dengan 4 buah CPU ($N = 4$)

$$t_{\text{par}} = (1-\alpha)T + \alpha T/N$$

$$= (1 - 0.6) * 10 \text{ menit} + 0.6 * 10 \text{ menit} / 4$$

$$= 4 \text{ menit} + 1.5 \text{ menit}$$

$$= 5.5 \text{ menit}$$

Contoh

- Speedup yang didapat

$$\begin{aligned}\text{speedup} &= t_{\text{seq}}/t_{\text{par}} \\ &= 10 \text{ menit} / 5.5 \text{ menit} \\ &= 1.82 \text{ kali}\end{aligned}$$

- Speedup maksimalnya

$$\begin{aligned}\text{speedupmax} &= 1 / (1-\alpha) \\ &= 1 / (1 - 0.6) \\ &= 1 / 0.4 \\ &= 2.5 \text{ kali}\end{aligned}$$

Kegagalan Amdahl Law

- Jika Amdahl Law benar, seharusnya berapa banyak CPU pun yang dikerahkan, maka speedup tidak akan banyak berubah (asimtotik)
- Tapi nyatanya ada super komputer yang terdiri dari banyak CPU yang performanya melebihi sebuah komputer CPU tunggal yang kuat
- Amdahl Law gagal untuk menjelaskan data empiris. Kenapa program paralel selalu melewati batas speedupnya.

Gustafson-Barsis Rebuttal

- Dua dekade setelah Amdahl Law dipublikasikan, Gustafson dan Barsis mengamati masalahnya dari sudut yang lebih tepat.
- Platform paralel lebih dari sekedar “mempercepat eksekusi program sekuensial”, tetapi mengakomodir masalah yang lebih besar.
- Jadi daripada melihat bagaimana program paralel melakukan sesuatu dibandingkan serialnya, kita seharusnya melihat bagaimana performa mesin sekuensial jika dia harus menyelesaikan masalah yang sama yang diselesaikan oleh mesin paralel.

Gustafson-Barsis Rebuttal

- Dengan mengasumsikan :
 - Ada aplikasi paralel yang membutuhkan waktu T untuk dieksekusi pada N buah CPU
 - Aplikasi ini menghabiskan sebanyak α bagian dari total waktunya secara paralel. Sisanya dikerjakan secara sekuensial
- Masalah yang sama dikerjakan pada mesin sekuensial akan membutuhkan waktu

$$t_{seq} = (1 - \alpha)T + N \cdot \alpha \cdot T$$

Gustafson-Barsis Rebuttal

- Speedup yang diperoleh sebesar

$$speedup = \frac{t_{seq}}{t_{par}} = \frac{(1 - \alpha)T + N \cdot \alpha \cdot T}{T} = (1 - \alpha) + N \cdot \alpha$$

- Dan efisiensinya

$$efficiency = \frac{speedup}{N} = \frac{1 - \alpha}{N} + \alpha$$

Contoh

- Sebuah program dijalankan pada komputer dengan 4 buah CPU menghabiskan waktu 10 menit. Setelah dianalisa, ternyata 60% bagian program tersebut dapat diparalelkan.
- Berapa waktu yang dibutuhkan untuk menjalankan program tersebut pada komputer dengan CPU tunggal?
- Berapa speedup yang diperoleh dan efisiensinya?

Contoh

- $T = 10$ menit
- $\alpha = 60\% = 0.6$
- Dengan 1 buah CPU

$$t_{seq} = (1-\alpha)T + N.\alpha.T$$

$$= (1 - 0.6) * 10 \text{ menit} + 4 * 0.6 * 10 \text{ menit}$$

$$= 4 \text{ menit} + 24 \text{ menit}$$

$$= 28 \text{ menit}$$

Contoh

- Speedup yang didapat

$$\begin{aligned}\text{speedup} &= t_{\text{seq}}/t_{\text{par}} \\ &= 28 \text{ menit} / 10 \text{ menit} \\ &= 2.8 \text{ kali}\end{aligned}$$

- Efisiensinya

$$\begin{aligned}\text{efisiensi} &= \text{speedup} / N \\ &= 2.8 / 4 \\ &= 0.7 \\ &= 70\%\end{aligned}$$

Contoh

- Dengan data yang sama dengan contoh Amdahl Law, didapatkan
 - Speedup = 2.8 kali
 - Efisiensi = 0.7 = 70%

Observed Speedup

- waktu eksekusi program serial dan program paralel tersebut adalah waktu nyata (*wall clock time*)
- tidak objektif untuk digunakan sebagai perbandingan karena waktunya tergantung pada :
 - kemampuan programmer mengimplementasikan program
 - pemilihan jenis compiler
 - switch compiler (contoh : menyalakan optimisasi atau tidak)
 - sistem operasi
 - filesystem yang menyimpan input data (NTFS, EXT4, dll)
 - waktu eksekusi (workload, traffic, dll)

Parallel Overhead

- Waktu yang dibutuhkan oleh task paralel untuk berkoordinasi tanpa melakukan pekerjaan komputasi sesungguhnya :
 - task startup time
 - synchronization
 - data communication
 - software overhead
 - task termination time

Parallel Programming Model

- Parallel Programming Model muncul sebagai abstraksi dari hardware dan arsitektur memori
- Ada beberapa yang umum digunakan :
 - Shared Memory
 - Threads
 - Message Passing
 - Data Parallel
 - Hybrid

- Model-model tersebut bukan model yang spesifik digunakan untuk mesin tertentu atau arsitektur memori tertentu.
- Secara teoritis, model-model ini bisa diimplementasikan pada hardware apapun.

Model Shared Memory pada Mesin Distributed Memory

- Kendall Square Research (KSR) ALLCACHE
- Memori terdistribusi secara fisik, tapi user melihatnya seperti single shared memory (global memory)
- Disebut virtual shared memory / distributed shared memory

Model Message Passing pada Mesin Shared Memory

- SGI Origin menggunakan arsitektur shared memory CC-NUMA dimana setiap task punya akses ke global memory.
- komunikasi data yang umum digunakan pada mesin SGI Origin tersebut adalah MPI, dengan mengirim dan menerima pesan seperti yang biasa digunakan pada arsitektur distributed memory.

Model 1 : Shared Memory

- antar task yang satu dengan yang lain saling berbagi alamat memori bersama. dibaca dan ditulis secara asynchronous
- beberapa mekanisme seperti locks/semaphores dapat digunakan untuk mengontrol akses ke memori bersama
- tidak ada “data ownership” sehingga programmer tidak perlu menyatakan secara eksplisit komunikasi data antar task
- sulit memahami dan mengatur lokalitas data

Model 1 : Shared Memory

- Implementasi :
 - kompiler menerjemahkan variabel pada program yang dibuat ke dalam alamat memori sesungguhnya yang bersifat global

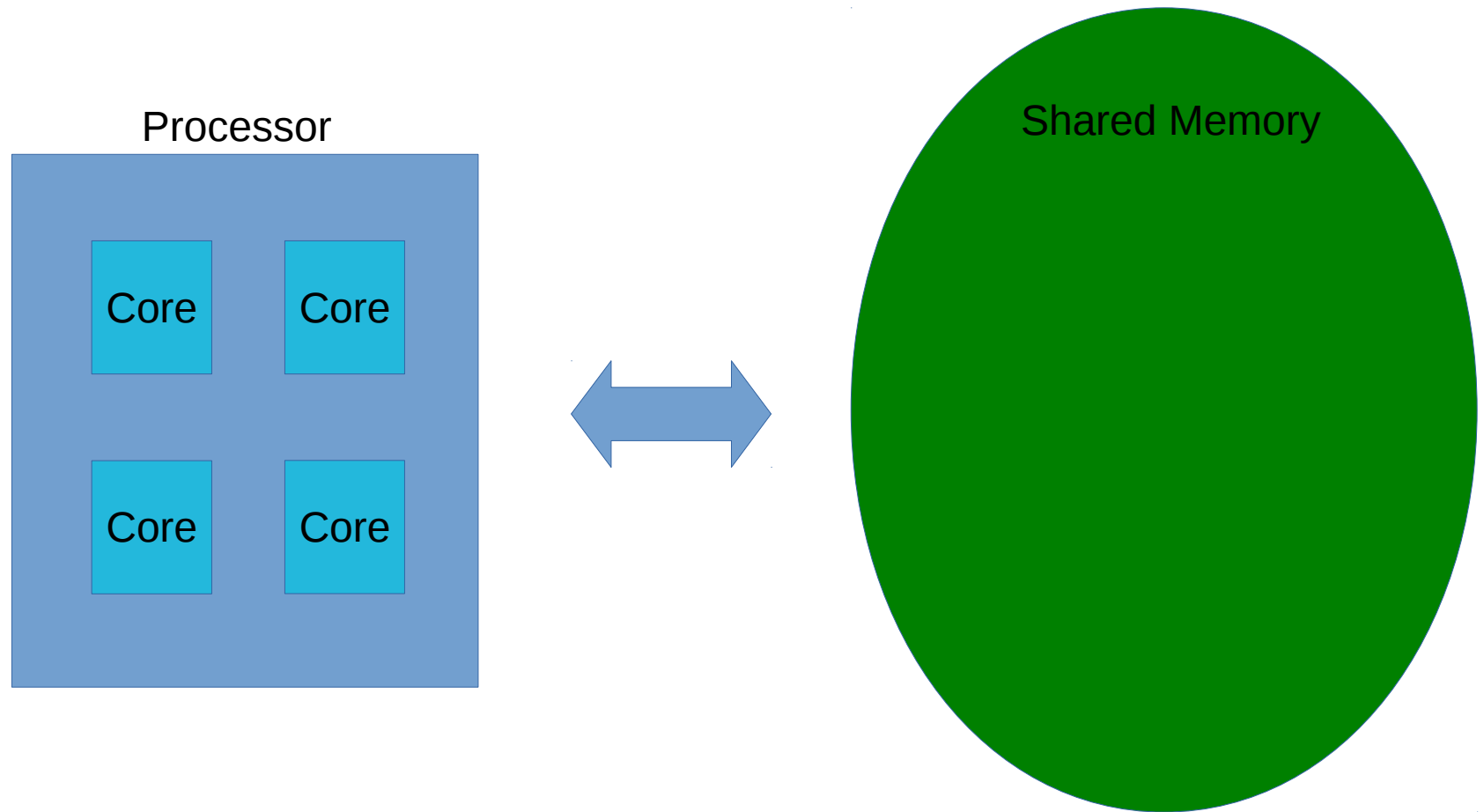
Model 2 : Threads

- sebuah proses dapat memiliki beberapa jalur eksekusi yang bersamaan (concurrent)
- main program dijalankan oleh OS secara serial.
- main program dapat menjalankan task (thread) yang terjadwal dan dijalankan oleh OS secara serentak
- setiap thread memiliki lokal data dan berbagi sumber daya bersama (shared memory)

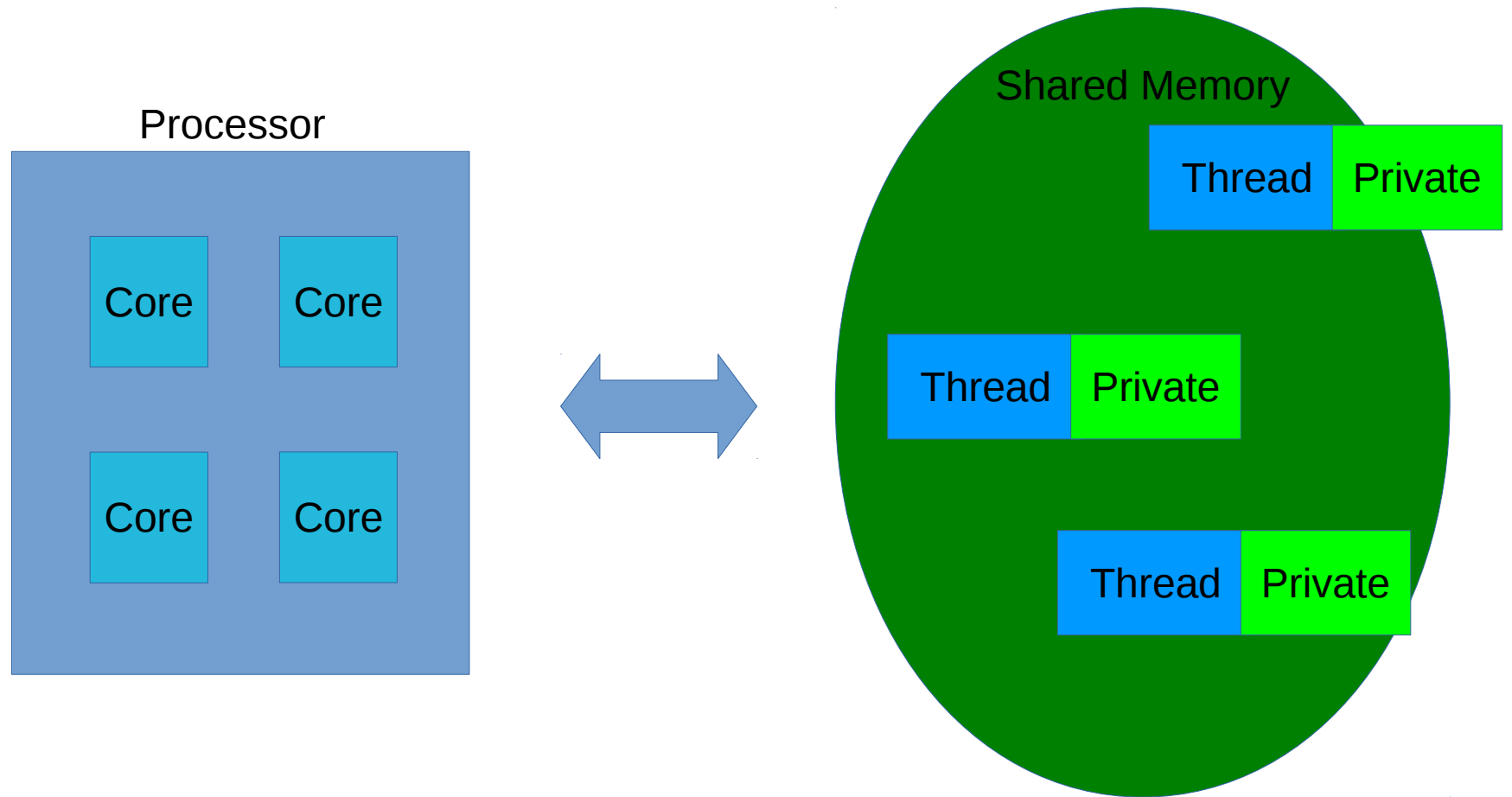
Model 2 : Threads

- thread berkomunikasi satu sama lain melalui global memory. karena itu dibutuhkan synchronization untuk memastikan tidak ada lebih dari satu thread yang mengupdate global memory pada satu waktu
- thread bisa dibuat dan dihancurkan, tapi main program tetap ada untuk menyediakan sumber daya bersama hingga program selesai dijalankan.

Threads



Threads



Model 2 : Threads

Implementasi :

- dari perspektif programming, implementasi thread terdiri dari
 - **library** yang dipanggil dari source code program paralel
 - sekumpulan **compiler directive** yang dimasukkan ke dalam source code program serial atau paralel
- pada keduanya, programmer bertanggung jawab untuk menentukan paralelisme yang dilakukan

Model 2 : Threads

- vendor hardware memiliki implementasi dari thread sendiri
- standarisasi implementasi tersebut mengerucut menjadi 2 macam implementasi threads :
 - POSIX Thread
 - OpenMP

POSIX Thread

- Berbasis library, koding paralel
- Bahasa C
- Dikenal sebagai pthreads
- Paralelisme eksplisit. membutuhkan kemampuan programmer untuk memperhatikan kode secara detail
- diatur dalam standard :
IEEE POSIX 1003.1c standard (1995)

Open MP

- Berbasis compiler directive, bisa menggunakan program serial
- Bahasa Fortran, C, C++
- Portabel, multiplatform
- Mudah dan sederhana dalam penggunaannya

Model 3 : Message Passing

- Setiap task menggunakan memori lokal pada waktu melakukan komputasi
- Task satu dan yang lain bertukar data dengan mengirimkan dan menerima paket data
- Data transfer biasanya membutuhkan operasi kooperatif dari tiap proses. (operasi send harus memiliki pasangan operasi receive)

Model 3 : Message Passing

Implementasi :

- Menggunakan library yang dimasukkan ke dalam penulisan source code.
- Programmer bertanggung jawab untuk menentukan paralelisme
- Pada arsitektur shared memory, implementasi MPI tidak menggunakan jaringan untuk komunikasi antar task, tapi menggunakan shared memory (salinan) untuk meningkatkan performa

Model 4 : Data Parallel

- Task paralel fokus pada melakukan operasi di sekumpulan data. data umumnya disusun dalam bentuk array atau matriks
- Sejumlah task bekerja bersamaan pada struktur data yang sama, tapi pada bagian data yang berbeda
- Task melakukan operasi yang sama pada data yang mereka kerjakan.

Model 4 : Data Parallel

Implementasi :

- Pada shared memory, semua task memiliki akses ke struktur data melalui memori global.
- Pada distributed memory, data strukturnya dibagi-bagi menjadi bagian kecil bernama “chunk” yang dikirimkan ke memori lokal masing masing task.

Model 5 : Hybrid

- Dua atau lebih model programming paralel dikombinasikan
- Yang umum : MPI digabungkan dengan thread (pthread atau openmp)
- Model hibrid yang umum lainnya adalah MPI digabungkan dengan data parallel. Implementasi data parallel pada arsitektur distributed memory menggunakan message passing sebagai cara komunikasinya.

Model Lainnya

- Model programming paralel tidak terbatas pada kelima model tersebut, tapi masih berkembang terus mengikuti perkembangan hardware dan software

Paralelisasi Otomatis vs Manual

- Pada awalnya, mendesain program paralel sepenuhnya pekerjaan manual :
 - time consuming
 - complex
 - error prone
- Beberapa tools dikembangkan untuk membantu programmer mengubah program serial ke dalam bentuk paralel (parallelizing compiler / pre-processor)

Parallelizing Compiler

- Fully Automatic
 - compiler menganalisis kode dan menemukan potensi untuk paralelisasi
 - mengidentifikasi penghambat (inhibitor) paralelisme dan menghitung apakah paralelisasinya meningkatkan performa atau tidak
 - mentarget loop untuk paralelisasi
- Programmer Directed
 - menggunakan compiler directive atau flags untuk memberitahu compiler bagian yang harus diparalelkan

Kelemahan Paralelisasi Otomatis

- bisa jadi hasilnya salah
- performa program paralel yang dihasilkan bisa lebih rendah dari program serialnya
- tidak sefleksibel paralelisasi manual
- terbatas pada bagian tertentu dari kode (biasanya loop)
- bisa tidak menghasilkan kode paralel jika hasil analisisnya mengatakan ada inhibitor atau kodenya terlalu kompleks
- kebanyakan dikembangkan dalam bahasa fortran

Paralelisasi Manual

Langkah-langkah mengembangkan program paralel :

- Memahami masalah yang ingin dipecahkan dan program (kode) yang ada
- Menentukan apakah masalahnya bisa diparalelkan atau tidak
- Mengatur data akses, komunikasi, dan sinkronisasi
- Mengidentifikasi hotspot, bottleneck, dan inhibitor dari algoritma yang dibuat

* note : algoritma paralel bisa berbeda sama sekali dengan algoritma serial meski untuk memecahkan masalah yang sama.

Metode PCAM

- Dipopulerkan oleh Ian Foster (1995)
- Metodenya masih relevan sampai sekarang, termasuk untuk platform multicore
- 4 langkah untuk mendesain program paralel :
 - Partitioning
 - Communication
 - Agglomeration
 - Mapping

Partitioning

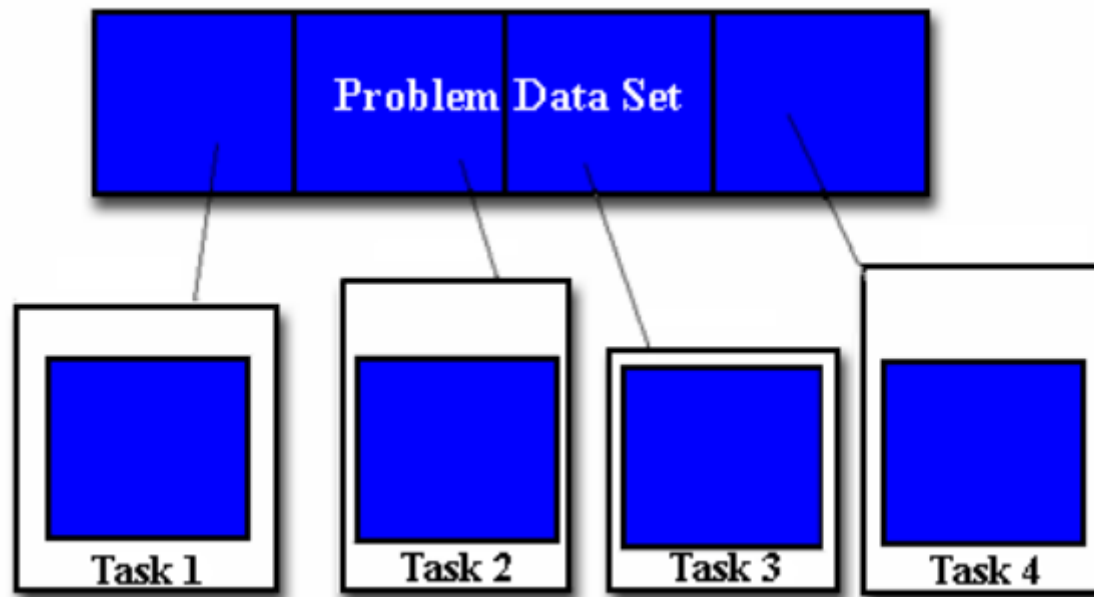
- Membagi masalah komputasi ke bagian-bagian kecil (task) yang bisa dikerjakan secara individu
- Tahap ini memunculkan paralelisme pada algoritmanya (jika ada)
- Pembagiannya bisa berdasarkan
 - data/domain decomposition
 - functional decomposition

Partitioning

- Tujuan tahap partitioning adalah untuk mengekspose kesempatan untuk melakukan eksekusi paralel
- Informasi praktis seperti “ada berapa prosesor yang akan digunakan” diabaikan dulu, masalah komputasi yang ada dibagi menjadi banyak task-task yang kecil (*fine grained decomposition of problems*)
- Kunci utama tahap partitioning adalah **mengidentifikasi dependensi**

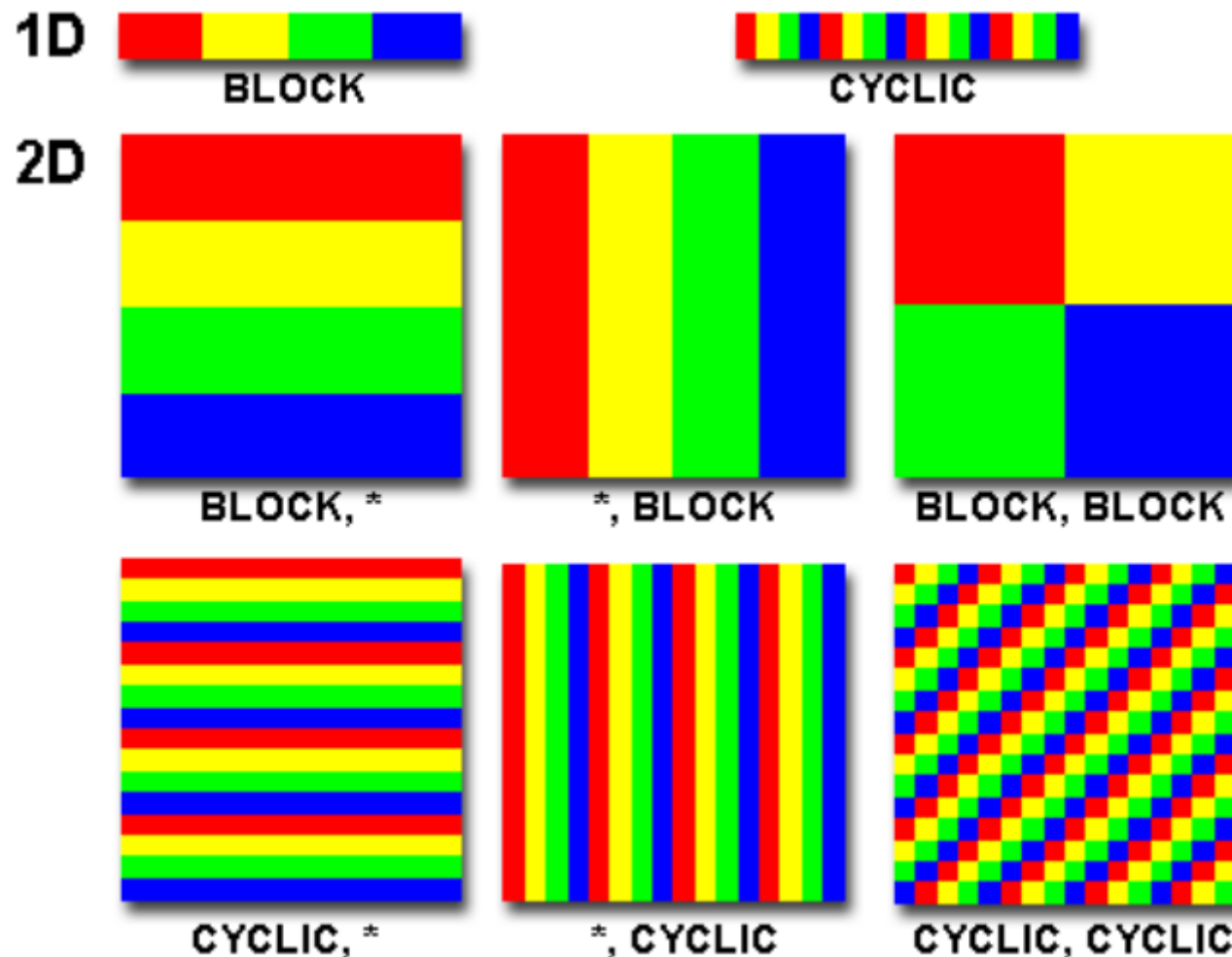
Domain Decomposition

- Data yang berkaitan dengan masalah yang ingin diselesaikan dipecah menjadi beberapa bagian



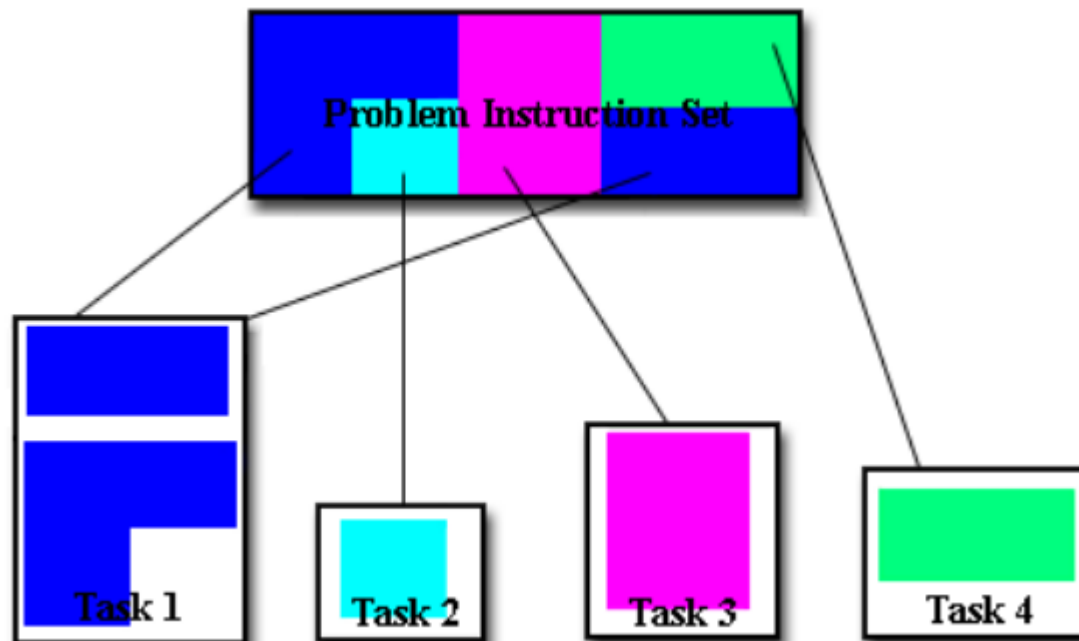
Domain Decomposition

- Beberapa cara untuk mempartisi data



Functional Decomposition

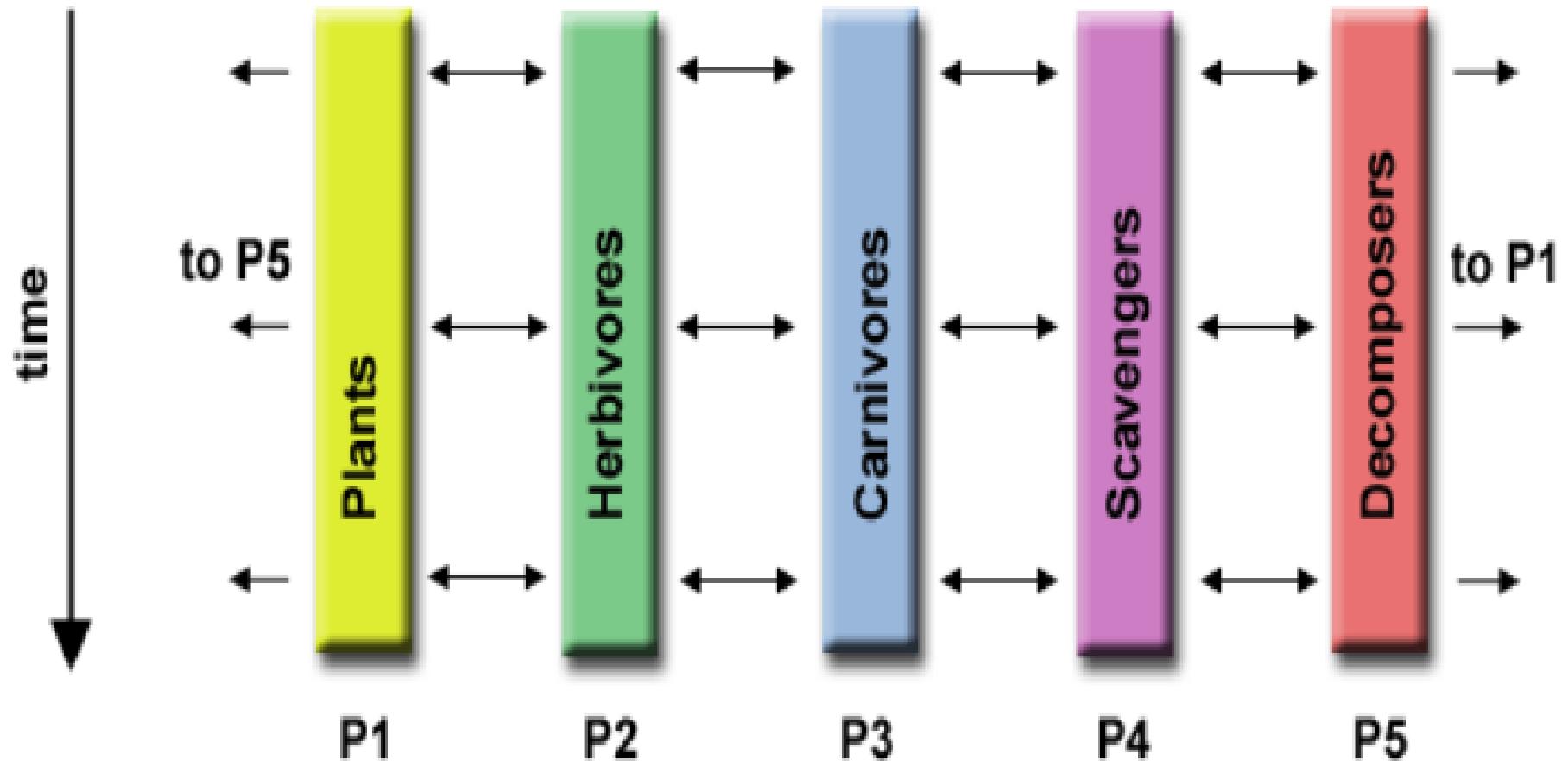
- Komputasi yang harus dilakukan dibagi menjadi beberapa bagian. Masalah dibagi berdasarkan pekerjaan yang harus diselesaikan. Setiap task melakukan bagian dari pekerjaan keseluruhan.



Contoh : Modeling Ekosistem

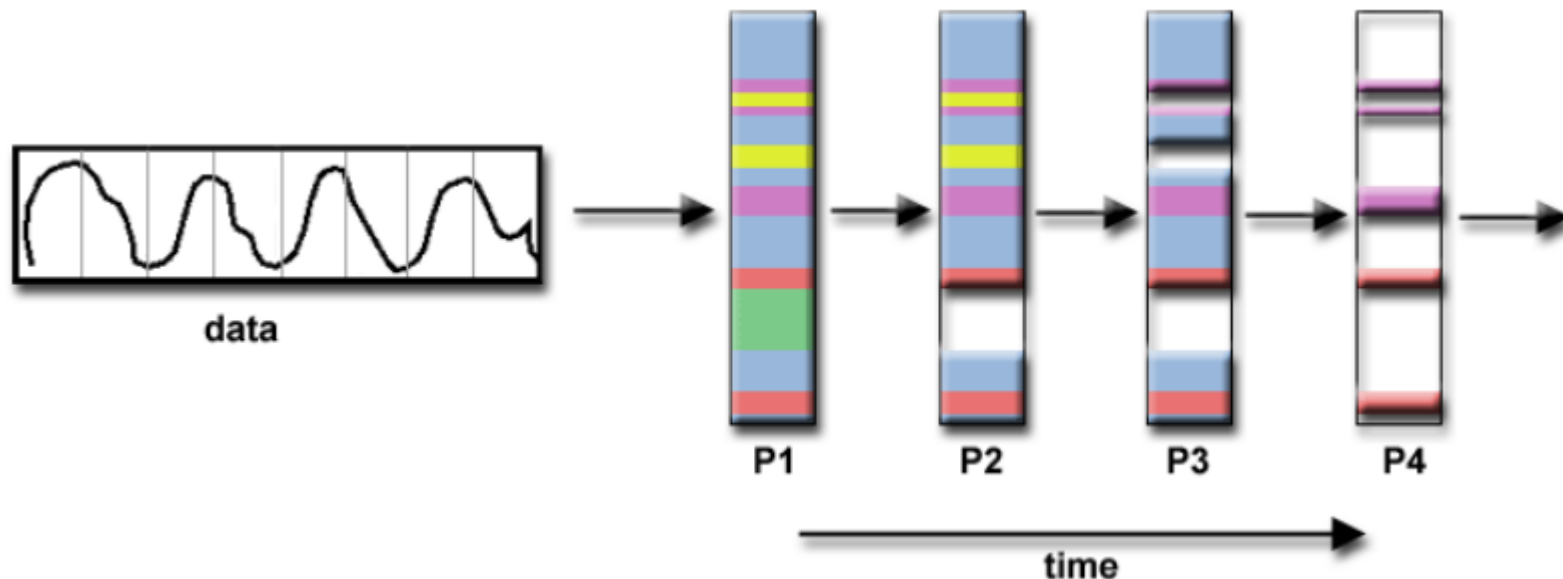
- Setiap program menghitung populasi dari sebuah grup
- Setiap grup berkembang tergantung grup tetangganya (prey - predator)
- Pada tiap langkah, masing masing proses menghitung kondisi terkini, mengkomunikasikan informasinya dengan grup tetangganya, dan menghitung kondisi pada langkah selanjutnya.

Contoh : Modeling Ekosistem



Contoh : Filter pada Sinyal Audio

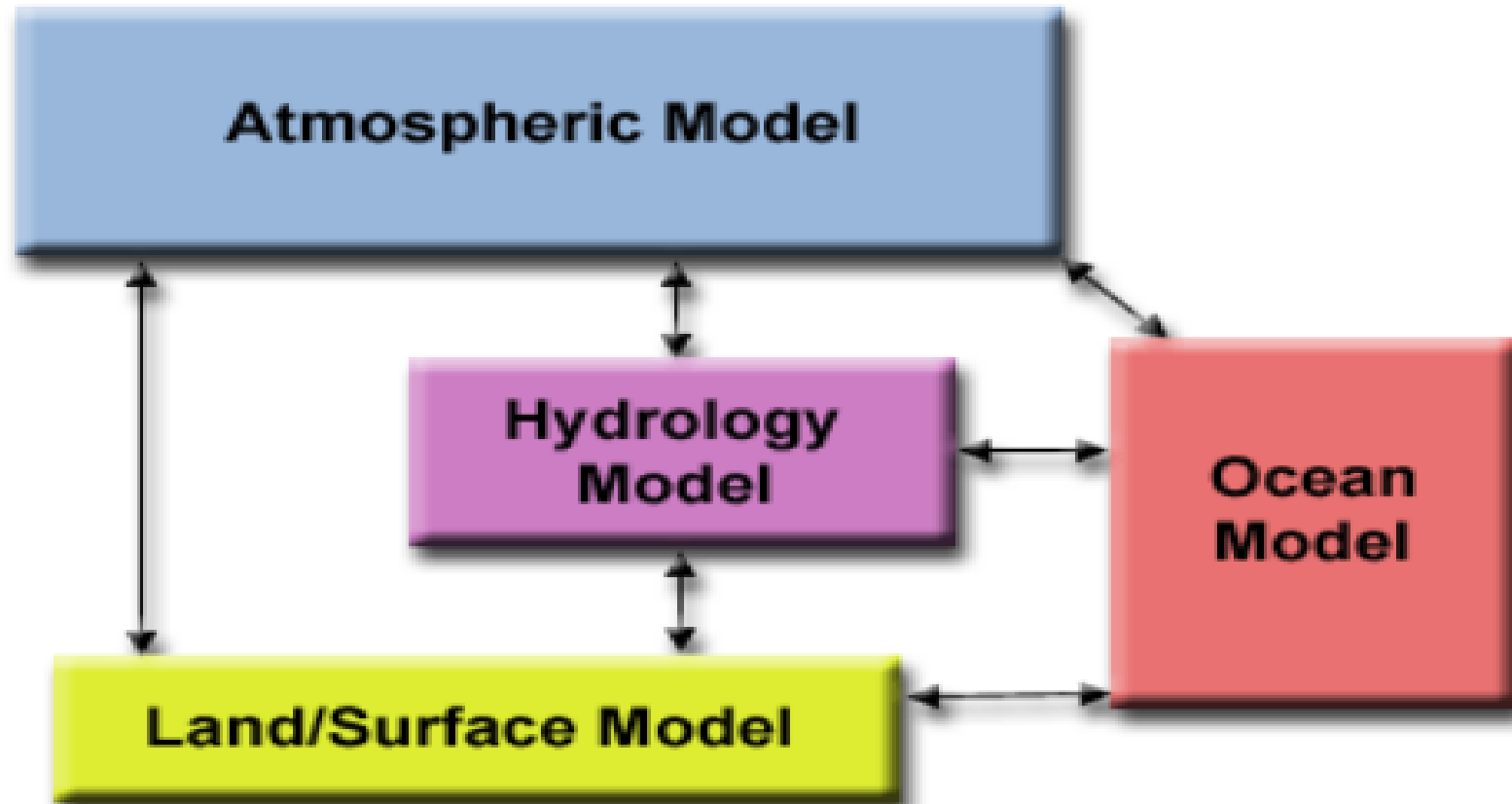
- Sinyal audio diproses melalui 4 buah filter yang merupakan proses terpisah. Prosesnya bertahap, sinyal harus melalui filter pertama sebelum masuk ke filter kedua. Pada saat data keempat masuk ke filter pertama, semua task bekerja



Contoh : Modeling Iklim

- Setiap komponen iklim dapat dipandang sebagai task yang berbeda
- Model atmosfer menghasilkan data kecepatan angin yang digunakan oleh model samudra. model samudra menghasilkan data suhu permukaan laut yang digunakan oleh model atmosfer
- Atmosfer menghasilkan data kelembaban udara yang digunakan oleh model hidrologi, dst.

Contoh : Modeling Iklim



Tugas

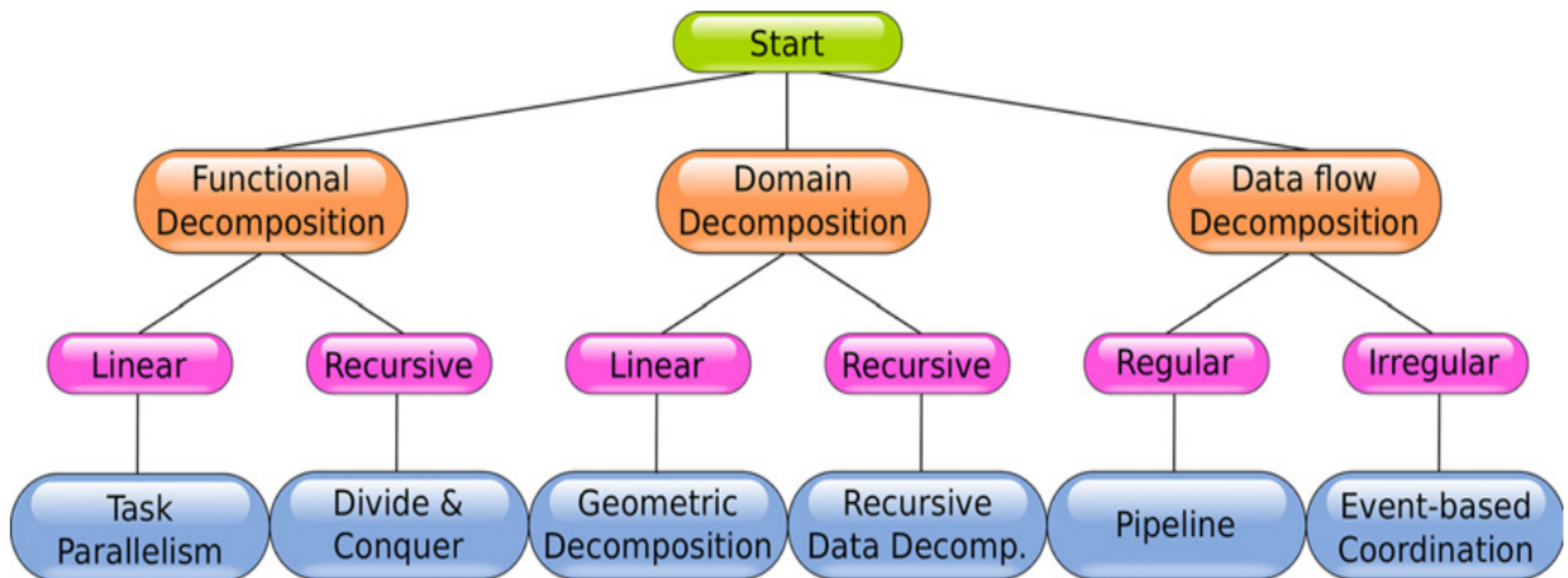
- Cari sebuah masalah dunia nyata yang dapat diparalelkan
- Bagi menjadi bagian bagian kecil
- Tentukan apakah itu domain/functional decomposition
- Cari sebuah algoritma/kode. pelajari kegunaan dan cara kerja algoritma tersebut.

Data Flow Decomposition

- Selain data/domain decomposition dan functional decomposition, ada satu cara lagi untuk mempartisi sebuah program.
- Kategori ini bisa diterapkan jika aplikasi digunakan untuk memproses data stream melalui beberapa tahap pemrosesan.
- Bisa dianggap sebagai ekstensi dari functional decomposition.
- Data stream bisa bertipe reguler ataupun ireguler

Pola Dekomposisi Algoritma

- Decision Tree untuk mendekomposisikan algoritma



Communication

- Pembagian data pada partitioning idealnya menyebabkan data saling independen.
- Tapi biasanya ada interdependensi antar task
- Volume data harus dikomunikasikan antar task yang interdependen ditentukan pada tahap ini
- Hasil dari tahap partitioning dan communication adalah **Task Dependency Graph**
 - Node = task
 - Edge = volume komunikasi

Agglomeration

- Mengurangi komunikasi antar prosesor
- Mengelompokkan task yang keterikatan datanya besar dalam sebuah grup
- Banyak grup yang dibuat sebaiknya setingkat di atas banyaknya node.

Mapping

- Grup yang sudah dibuat, diassign (map) ke dalam sebuah node komputasi (proses)
- Tujuannya :
 - load balance
 - mengurangi overhead komunikasi dengan memetakan grup dengan komunikasi besar ke dalam node yang sama
- Mapping dari proses ke prosesor diserahkan kepada OS, Compiler, atau Hardware

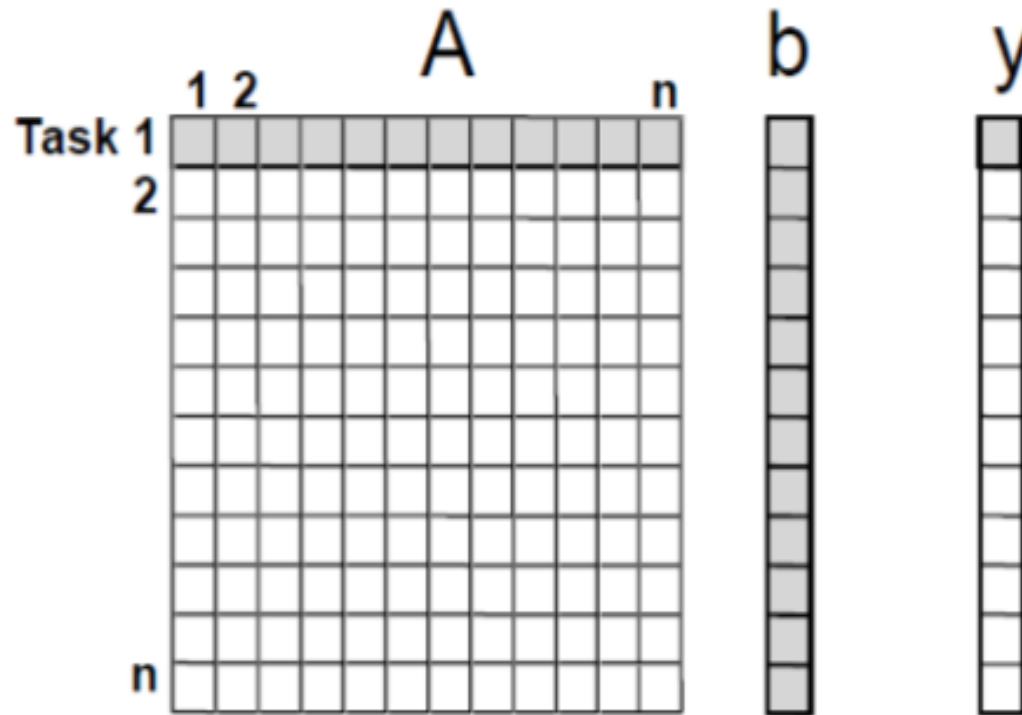
Mapping

- Meletakkan grup yang berkaitan (*cooperating threads*) dalam prosesor yang sama akan memaksimalkan lokalitas, data sharing, serta meminimalkan biaya komunikasi dan sinkronisasi
- Meletakkan grup yang tidak berkaitan pada prosesor yang sama untuk menggunakan mesin lebih efisien

Contoh : Perkalian Matriks - Vektor

$$A * b = y$$

- A dan b adalah matriks dan vektor yang padat (dense)



Contoh : Perkalian Matriks - Vektor

- Problem : menghitung vektor $\mathbf{y} = \mathbf{A} * \mathbf{b}$
- Task :
 - menghitung $\mathbf{y}[i]$ hanya membutuhkan baris ke- i dari \mathbf{A} dan vektor \mathbf{b}
 - menghitung $\mathbf{y}[i]$ dapat dianggap sebagai task
- Catatan :
 - ukuran task seragam
 - tidak ada ketergantungan antar task satu dan yang lainnya
 - semua task butuh \mathbf{b}

Contoh : Perkalian Matriks - Vektor

- Misal $n = 2$, kita memiliki problem berikut

a11	a12	x	b1	=	y1
a21	a22		b2		y2

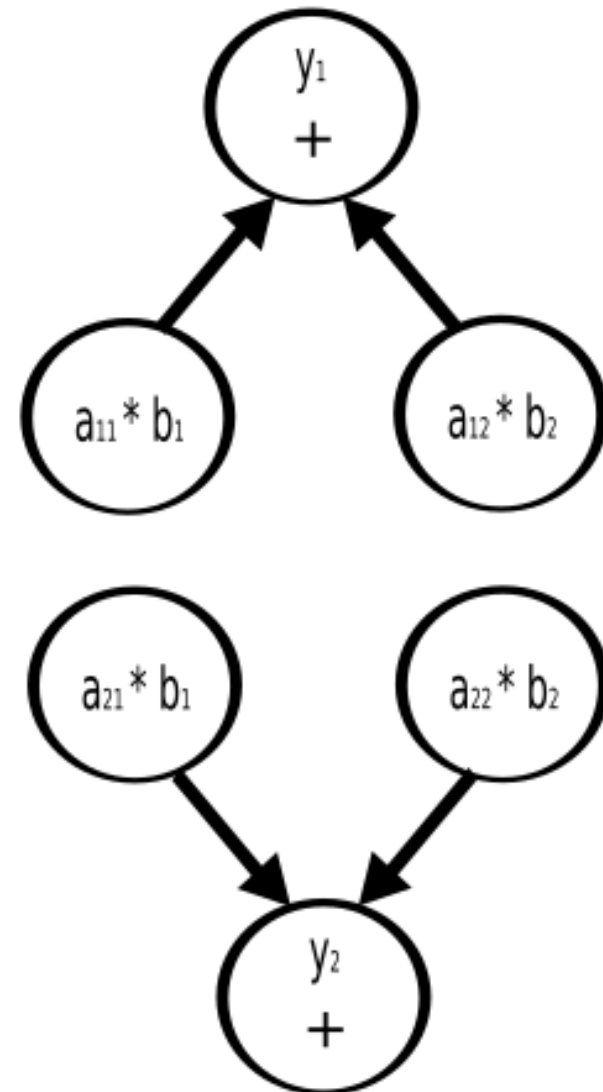
$$y1 = a11*b1 + a12*b2$$

$$y2 = a21*b1 + a22*b2$$

- Partitioning
 - Problem di atas dapat dibagi sekecil-kecilnya menjadi 8 bagian, yaitu tiap data berdiri sendiri (data decomposition)
 - Atau maksimal menjadi 6 kalkulasi (functional decomposition)

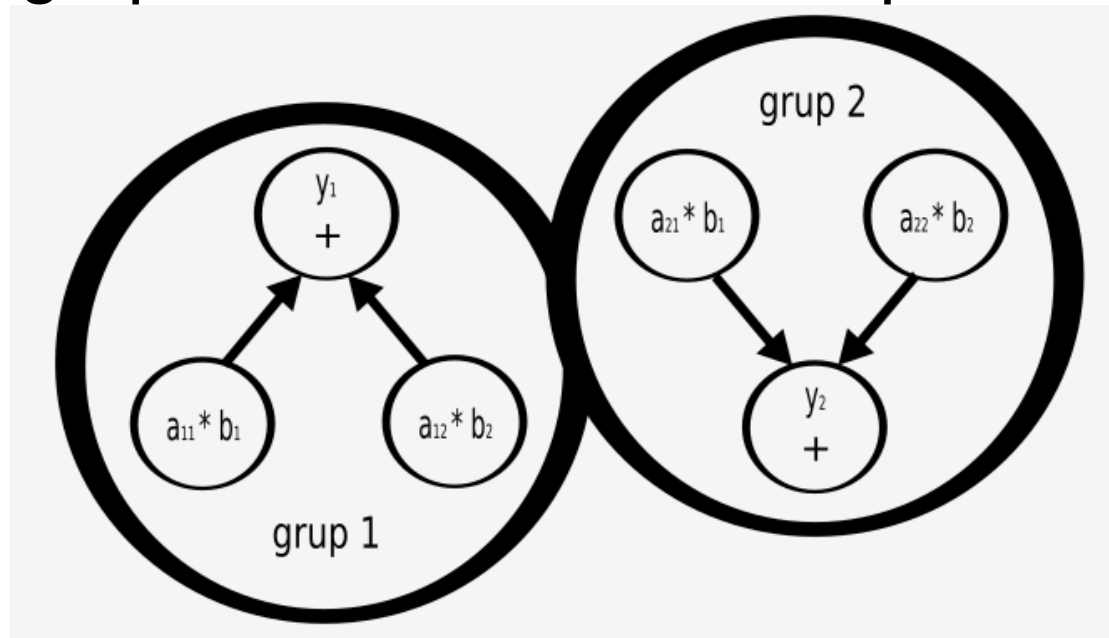
Contoh : Perkalian Matriks - Vektor

- Communication
 - Jika kita membagi problem tersebut berdasarkan datanya akan dibahas belakangan (*task interaction graph*)
 - Jika kita membagi problem tersebut menjadi 6 task dengan functional decomposition, maka didapat task dependency graph berikut



Contoh : Perkalian Matriks - Vektor

- Agglomeration
 - Untuk kasus ini, terlihat jelas keterkaitan dari task yang ada sehingga kita bisa buat jadi 2 grup
- Mapping
 - Tiap grup task dialokasikan ke proses



Contoh : Perkalian Matriks - Vektor

- Tidak ada communication antar task
(*embarrassingly parallel*)
- **Embarrassingly parallel / perfectly parallel** = masalah komputasi paralel dimana usaha untuk membagi masalah menjadi task paralel sangat sederhana. Biasanya antar task membutuhkan sedikit komunikasi atau tidak sama sekali.

Contoh : Database Query

- query : Model = "Civic" **and** Year = "2001" **and** (Color = "Green" **or** Color = "White")

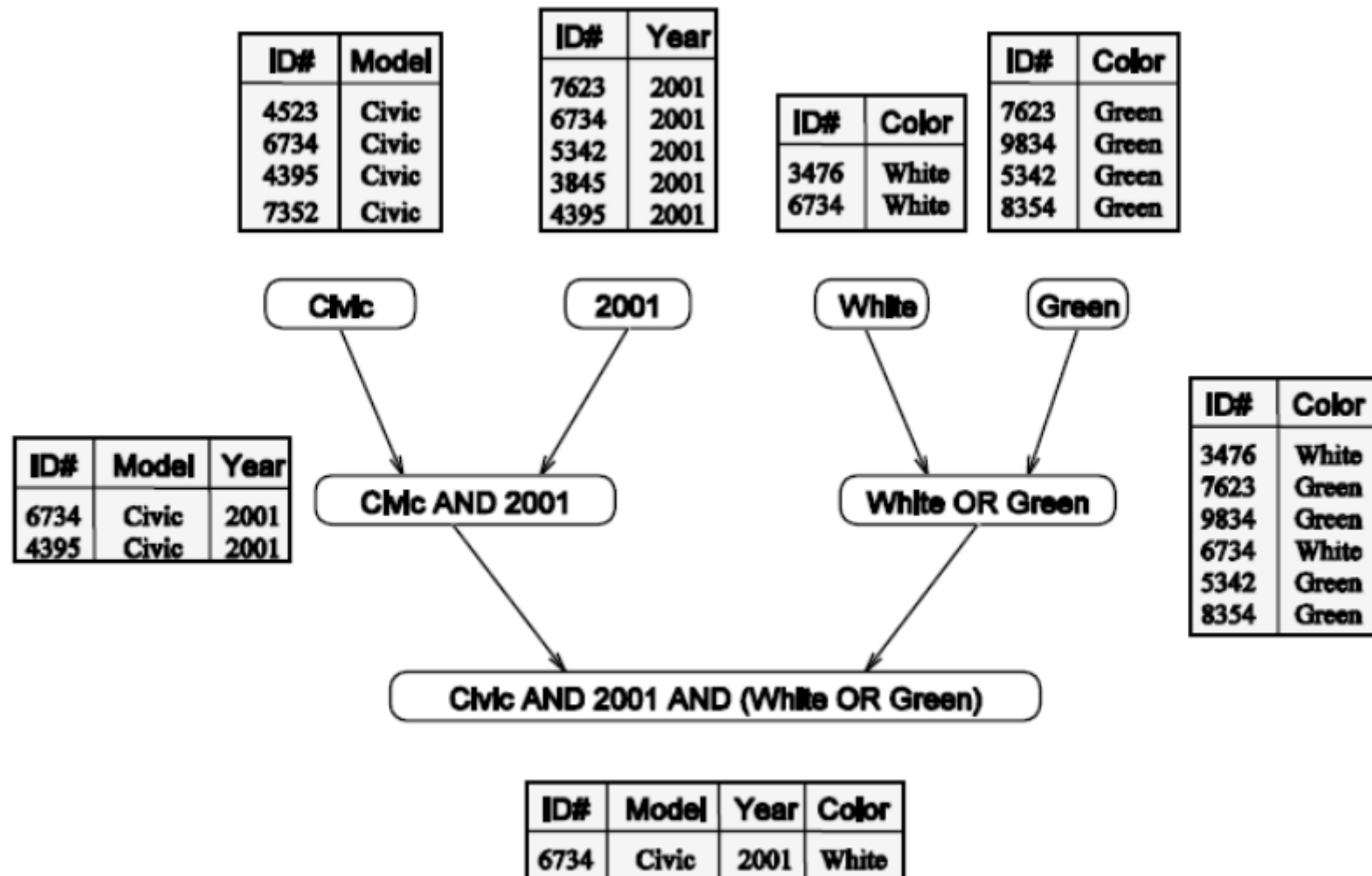
ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	\$18,000
3476	Corolla	1999	White	IL	\$15,000
7623	Camry	2001	Green	NY	\$21,000
9834	Prius	2001	Green	CA	\$18,000
6734	Civic	2001	White	OR	\$17,000
5342	Altima	2001	Green	FL	\$19,000
3845	Maxima	2001	Blue	NY	\$22,000
8354	Accord	2000	Green	VT	\$18,000
4395	Civic	2001	Red	CA	\$17,000
7352	Civic	2002	Red	WA	\$18,000

Contoh : Database Query

- Problem : mencari data (row) yang memenuhi **semua** kriteria pada query
- Task :
 - sejumlah elemen yang memenuhi kriteria tertentu
- Task Dependency Graph
 - Node = task
 - Edge = output dari task yang satu jadi input dari task yang lain

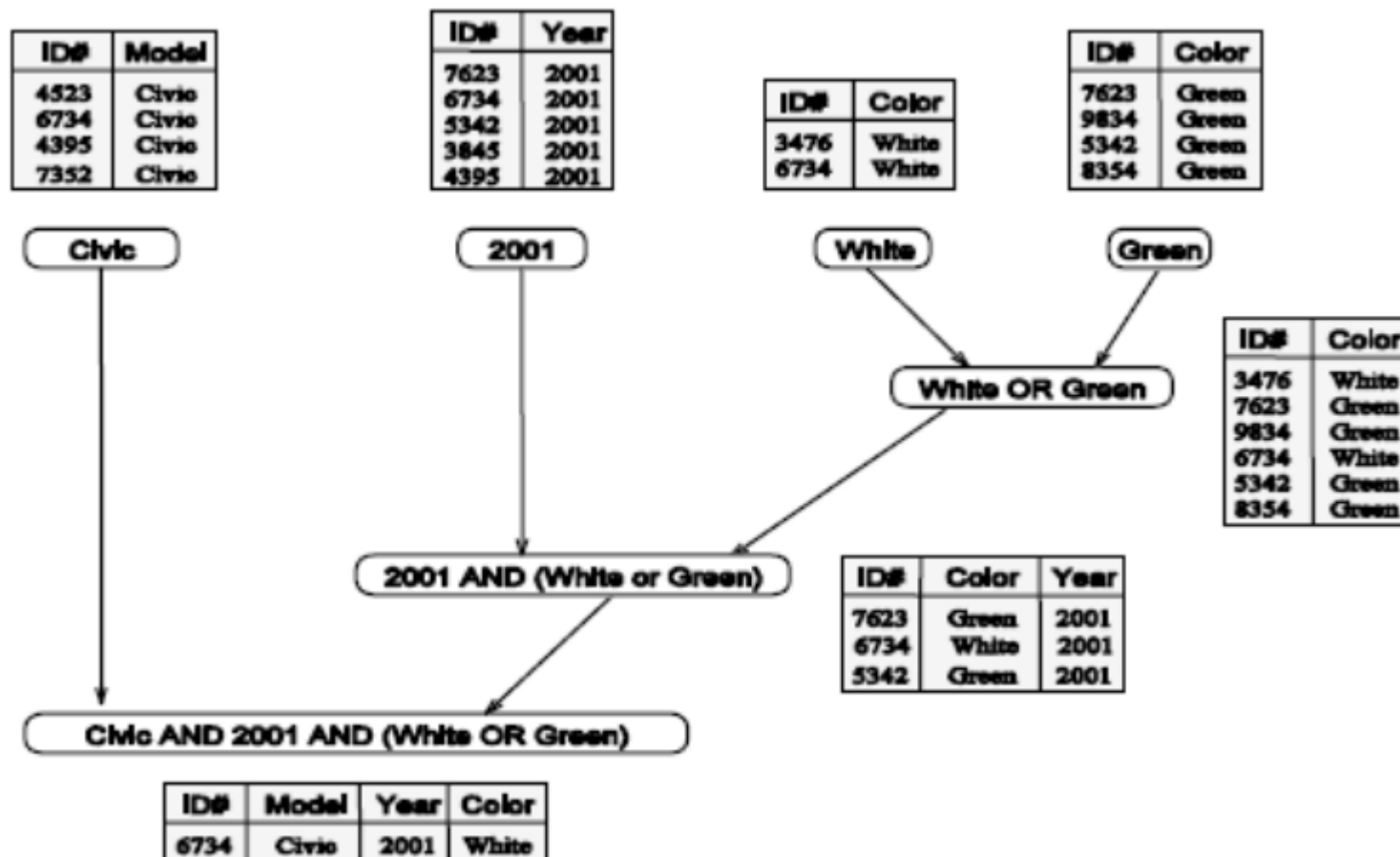
Task Dependency Graph

$$A = (p \text{ and } q) \text{ and } (r \text{ or } s)$$



Task Dependency Graph

$B = p \text{ and } (q \text{ and } (r \text{ or } s))$

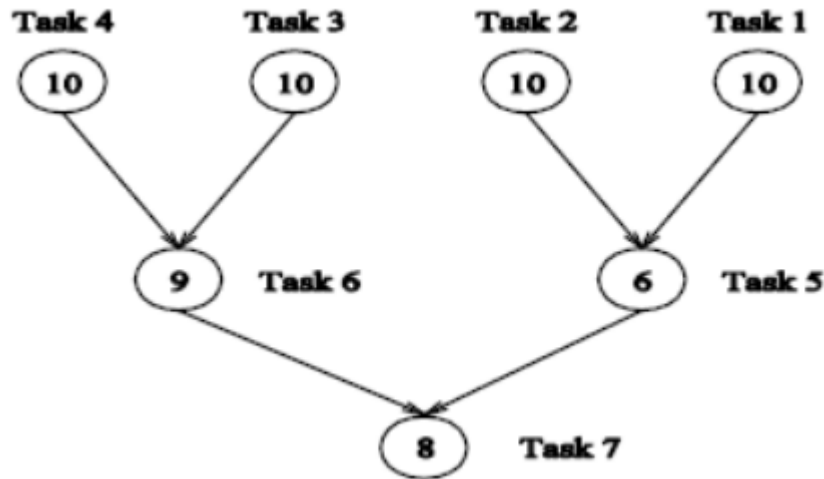


Derajat Keserentakan

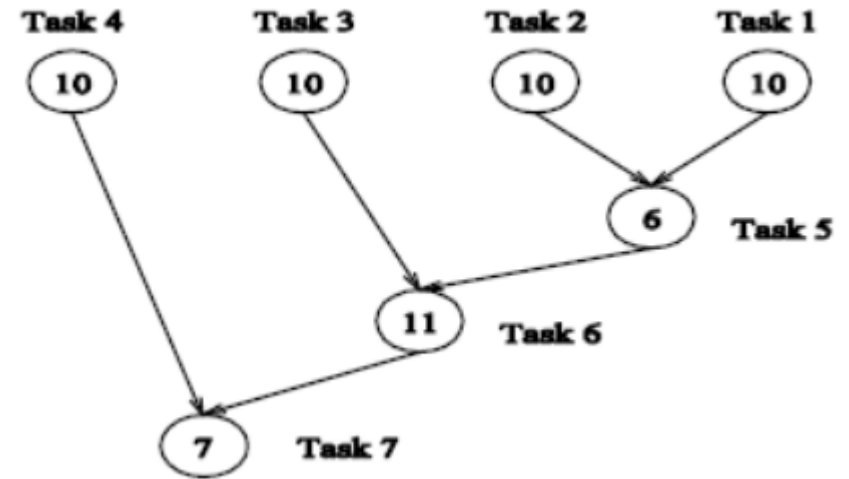
- *Degree of Concurency* = Banyaknya task yang bisa dieksekusi secara paralel
 - maximum degree of concurency = banyak task serentak yang paling besar pada bagian manapun dari eksekusi
 - average degree of concurency = rata-rata dari banyaknya task yang bisa dieksekusi secara paralel
- **Derajat keserentakan** berbanding terbalik dengan **granularitas task**

Critical Path

- *Critical Path* = path terpanjang dari node awal (node tanpa edge masuk) dan node akhir (node tanpa edge keluar)
- Critical Path Length = jumlah beban maksimal tiap node pada critical path
- average degree of concurrency =
$$\frac{\text{total amount of work}}{\text{critical path length}}$$



Graf A



Graf B

- Critical path length =

27

34

- Average degree of concurrency =

$$63/27 = \mathbf{2,33}$$

$$64/34 = \mathbf{1,88}$$

Task Interaction Graph

- Antar task biasanya berkomunikasi input, output, atau data yang diprosesnya.
- Task Interaction Graph
 - Node = task
 - Edge = interaksi / pertukaran data
- Task Dependency vs Task Interaction
 - Task dependency graph merepresentasikan keterkaitan **kontrol** antar task
 - Task interaction graph merepresentasikan keterkaitan **data** antar task

Contoh : Perkalian Matriks Jarang (Sparse Matrix)

- baris ke- i dari matriks A dan elemen $b[i]$ diassign ke task ke- i
- setiap task menghitung elemen y (task ke- i menghitung $y[i]$)

