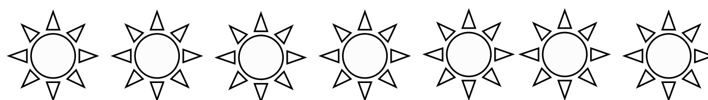
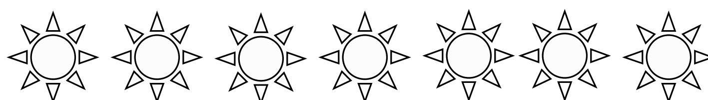




2024 M. Damrudi

مقدمه

نیاز به محاسبات موازی

- ❖ پردازش داده ها یا انجام محاسبات بسیار زیاد در مدت زمان قابل قبول
- ❖ کامپیوتر موازی (parallel computer)
  - ➢ یک کامپیوتر با تعداد زیادی از واحدهای پردازشی یا پردازنده ها
- ❖ حل مسأله با استفاده از یک کامپیوتر موازی:
  - ➢ تقسیم مسأله به تعدادی زیرمسأله
  - ➢ حل همزمان زیرمسائل با حل هر زیرمسأله به وسیله یک پردازشگر جداگانه
  - ➢ ترکیب نتایج برای به دست آوردن راه حل مسأله اصلی
- ❖ الگوریتم موازی
  - ➢ روش حل یک مسأله به منظور اجرا بر روی یک کامپیوتر موازی

مقدمه

مدلهای محاسباتی

- هر کامپیوتر چه ترتیبی چه موازی با اجرای دستورالعمل ها روی داده عمل می کنند.
  - جریان دستورالعمل ها در هر مرحله به کامپیوتر می گوید چه کاری انجام دهد.
  - جریان داده ها از این دستورالعمل ها تاثیر می گیرند.
- طراحی الگوریتم های موازی نیازمند داشتن درک درستی از مدل های محاسباتی است.
- چهار دسته از معماری کامپیوترها که توسط Flynn در ۱۹۶۶ ارائه شده است:
- یک دستورالعمل، یک داده (SISD)
- چند دستورالعمل، یک داده (MISD)
- یک دستورالعمل، چند داده (SIMD)
- چند دستورالعمل، چند داده (MIMD)

مقدمه

یک دستورالعمل، یک داده (SISD)

- دسته بندی های ذکر شده بر اساس تعداد instructions و dataیی که پردازش می شوند است.
- این مدل توسط John von Neumann در اواخر ۱۹۴۰ ارائه شد.
- مدل سریال یا ترتیبی نام دارد.
- الگوریتم روی کامپیوترهای SISD هیچ پردازش موازی ندارد.
- برای پردازش موازی به بیشتر از یک پردازنده نیاز است.
- مثال: جمع n عدد
  - n بار دسترسی به حافظه برای دسترسی به n عدد به ترتیب
  - n-1 عمل جمع به ترتیب
  - O(n)

مقدمه

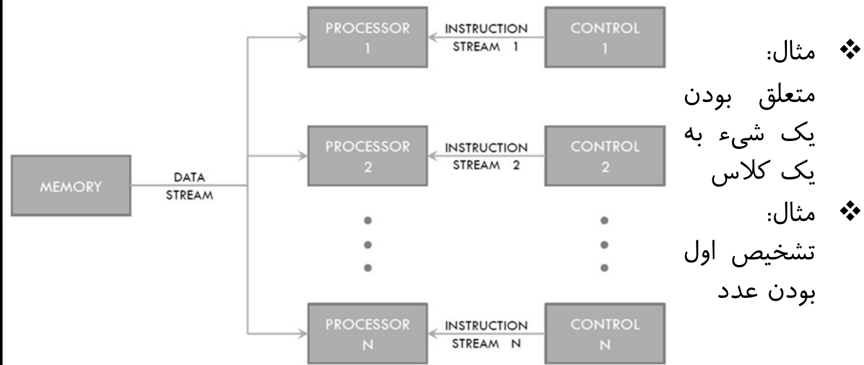
چنددستورالعمل، یک داده (MISD)

❖ N پردازنده که هر کدام واحد کنترل خود را دارند، یک واحد حافظه را به اشتراک گذاشته اند. در بسیاری از برنامه های کاربردی می تواند مفید باشد.

❖ در هر مرحله، داده ای که از حافظه دریافت می شود توسط تمام پردازنده ها همزمان مورد پردازش قرار می گیرند. هر پردازنده براساس دستوراتی که از واحد کنترل دریافت می کند، عمل می کند.

❖ مثال: متعلق بودن یک شیء به یک کلاس

❖ مثال: تشخیص اول بودن عدد



مقدمه

یک دستورالعمل، چند داده (SIMD)

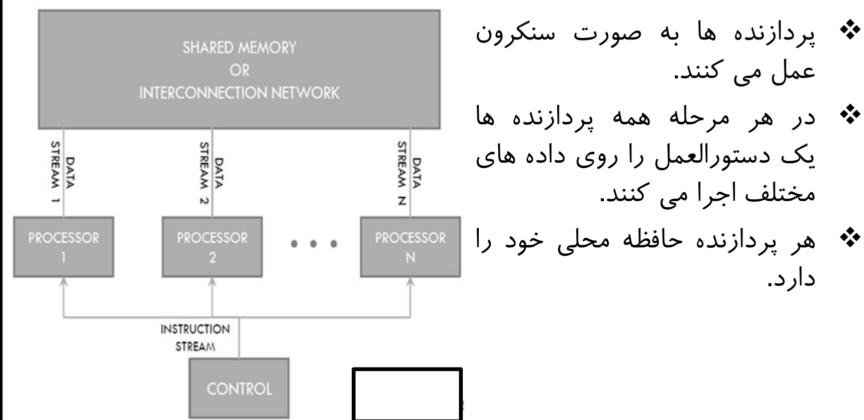
❖ در این کلاس، یک کامپیوتر موازی شامل N پردازنده یکسان است.

❖ همه پردازنده ها تحت کنترل یک جریان دستورالمل ساده که توسط واحد کنترل مرکزی ارائه می شود عمل می‌کنند. N جریان داده برای N پردازنده وجود دارد.

❖ پردازنده ها به صورت سنکرون عمل می کنند.

❖ در هر مرحله همه پردازنده ها یک دستورالعمل را روی داده های مختلف اجرا می کنند.

❖ هر پردازنده حافظه محلی خود را دارد.



مقدمه

یک دستورالعمل، چند داده (SIMD)

- ❖ دستورالعمل می تواند ساده (جمع یا مقایسه دو عدد) یا پیچیده (ادغام دو لیست) باشد.
- ❖ داده می تواند ساده (یک داده)، یا پیچیده (چند عدد) باشد.
- ❖ در این کلاس، یک کامپیوتر موازی شامل N پردازنده یکسان است.
- ❖ همه پردازنده ها تحت کنترل یک جریان دستورالعمل ساده که توسط واحد کنترل مرکزی ارائه می شود عمل می‌کنند. N جریان داده برای N پردازنده وجود دارد.
- ❖ ارتباط بین پردازنده ها برای انتقال داده ها در کامپیوترهای SIMD به دو روش است:
  - ➢ shared memory
  - ➢ interconnection network

مقدمه

Shared-Memory (SM) SIMD Computers

- ❖ این گروه با نام Parallel Random-Access Machine (PRAM) model (ماشین موازی با دستیابی تصادفی) شناخته می شود.
- ❖ ارتباط میان پردازنده ها از طریق حافظه مشترک انجام می شود.
- ❖ اگر پردازنده i بخواهد داده ای را به پردازنده j ارسال کند، در دو مرحله انجام می شود.
  - ➢ پردازنده i داده را درحافظه مشترک در یک محل شناخته شده برای پردازنده j می‌نویسد.
  - ➢ پردازنده j داده را از محل مورد نظر می خواند.
- ❖ کامپیوترهای shared-memory SIMD بر اساس نحوه دسترسی همزمان به یک مکان یکسان به چهار دسته تقسیم می شوند.
- ❖ Exclusive-Read, Exclusive-Write (EREW)
- ❖ Concurrent-Read, Exclusive-Write (CREW)
- ❖ Exclusive-Read, Concurrent-Write (ERCW)
- ❖ Concurrent-Read, Concurrent-Write (CRCW)

مقدمه

Shared-Memory (SM) SIMD Computers

- EREW: تمام پردازنده ها اجازه خواندن و نوشتن همزمان در یک خانه حافظه را ندارند (مدل ساده و ارزان).
- CREW: چند پردازنده اجازه خواندن از یک خانه حافظه را همزمان دارند اما به طور همزمان اجازه نوشتن در یک خانه حافظه را ندارند.
- ERCW: چند پردازنده اجازه نوشتن در یک خانه حافظه را همزمان دارند اما به طور همزمان اجازه خواندن از یک خانه حافظه را ندارند.
- CRCW: اجازه خواندن و نوشتن همزمان در یک خانه حافظه وجود دارد.
- نوشتن همزمان در یک مکان یکسان می تواند باعث بروز مشکل (conflict write) شود. سیاستهای مختلفی برای حل این مشکل وجود دارد:
  - پردازنده با کوچکترین شماره مجاز به نوشتن است و دیگر پردازنده ها اجازه نوشتن را ندارند.
  - اگر کمیت‌هایی که قرار است نوشته شوند با یکدیگر برابر باشند، همه پردازنده‌ها مجاز به نوشتن هستند. در غیراینصورت هیچ‌یک از پردازنده‌ها اجازه نوشتن را ندارند.
  - مجموع کمیت هایی که قرار است نوشته شوند، ذخیره گردد.

مقدمه

Shared-Memory (SM) SIMD Computers

- مثال: جستجوی x در یک فایل نامرتب بسیار بزرگ شامل n ورودی متمایز
- استفاده از مدل EREW با N پردازنده به طوری که $N \leq n$:
- مرحله۱: انتشار مقدار x به همه پردازنده ها در زمان $O(\log N)$
  - خواندن x توسط P_1 و ارسال آن به P_2،
  - ارسال همزمان x از P_1 و P_2 به ترتیب به P_3 و P_4،
  - ارسال همزمان x از P_1، P_2، P_3 و P_4به ترتیب به P_5، P_6، P_7 و P_8
  - تا زمانیکه تمام پردازنده ها x را داشته باشند ادامه می یابد. (فصل بعد)
- مرحله۲: جستجوی همزمان در زیرفایل‌ها به وسیله پردازنده‌ها در زمان $O(n/N)$
  - پردازنده P_1، n/N داده اول را جستجو می‌کند.
  - پردازنده P_2، n/N داده دوم را جستجو می‌کند.
  - و ...

مقدمه

Shared-Memory (SM) SIMD Computers

- ابتدا F مقدار false دارد. هر پردازنده‌ای که داده را پیدا کند F را true می‌کند.
- در هر مرحله تمام پردازنده ها F را چک می کنند. اگر یکی از پردازنده ها موفق به یافتن x شود، مقدار F را برابر با True قرار داده و در نتیجه عمل جستجو در همه پردازنده ها پایان می پذیرد. اگر true باشد الگوریتم خاتمه می یابد.
- انتشار مقدار F در هر مرحله به پردازنده ها log N مرحله است.
- در بدترین حالت مدل EREW به log N+(n/N)log N مرحله نیاز است.
- استفاده از مدل CREW با N پردازنده
- مرحله۱: خواندن مقدار x توسط همه پردازنده ها همزمان O(1)
- مرحله۲: جستجوی همزمان در زیرفایل‌ها به وسیله پردازنده‌ها در زمان O(n/N)
- مقدار F همزمان توسط تمام پردازنده ها خوانده می شود.
- ✓ در صورتیکه داده ها متمایز(متفاوت) نباشد چه مشکلی پیش می آید؟ چگونه می توان این مشکل را حل کرد.

مقدمه

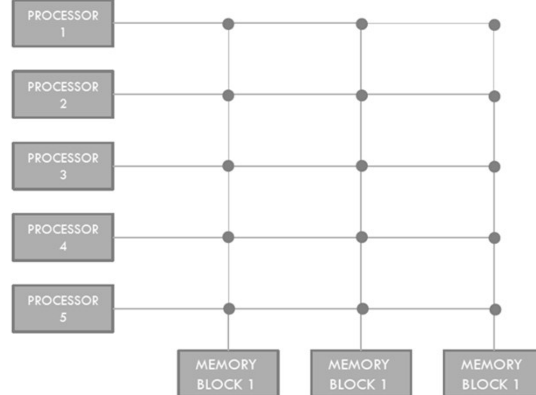
Shared-Memory (SM) SIMD Computers

- برای ایجاد مسیر از حافظه به محلی در حافظه مداراتی نیاز است.
- اگر حافظه شامل M محل باشد، هزینه مدارها f(M) است.
- اگر N پردازنده حافظه را به اشتراک بگذارند، هزینه مدارها N×f(M) است.
- برای N و M های بزرگ، هزینه مدارها بسیار زیاد می شود.
- راه های مختلفی برای حل این مشکل ارائه شده است:
  - روش اول:
  - تقسیم بندی حافظه اشتراکی به R بلوک با اندازه‌های مساوی (هر کدام M/R)
  - روش دوم:
  - توزیع N مکان حافظه در میان N پردازنده به طور مساوی (M/N)

مقدمه

Shared-Memory (SM) SIMD Computers

- تقسیم بندی حافظه اشتراکی به R بلوک با اندازه های مساوی (هر کدام M/R)
- در هر لحظه، فقط یک پردازنده می تواند به یک بلوک دسترسی داشته باشد.
- N+R خط دوطرفه برای دسترسی هر پردازنده به هر بلوک حافظه در هر زمان
- هزینه کل مدارها R×f(M/R)، البته N×R سوئیچ نیز نیاز است. N=5، R=3

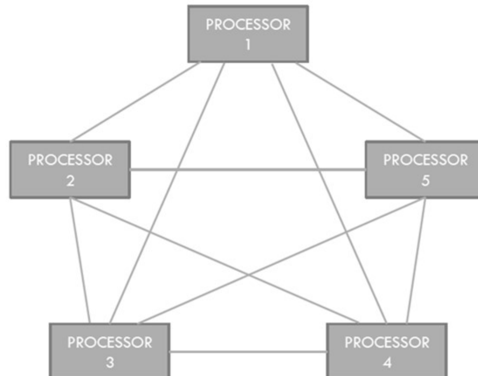


مقدمه

Interconnection Network SIMD Computers

- توزیع M محل حافظه در میان N پردازنده به طور مساوی (M/N)
- متصل نمودن هر زوجی از پردازنده ها از طریق یک خط ارتباطی مستقیم
- انتقال سریع اطلاعات بین دو پردازنده به دلیل ارتباط مستقیم

Fully Connected Network for N=5



- نحوه آدرس دهی:
  - Diameter برابر با ۱
  - Degree برابر با O(N)
- هزینه: تعداد اتصالات بسیار زیاد است. N(N-1)/2 خط ارتباطی دو طرفه برای این مدل $O(N^2)$
  - راه حل: استفاده از شبکه های اتصالی ساده تر

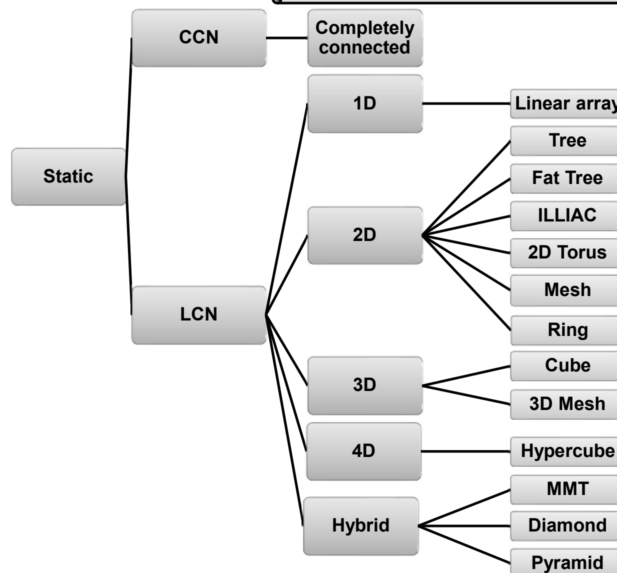
مقدمه

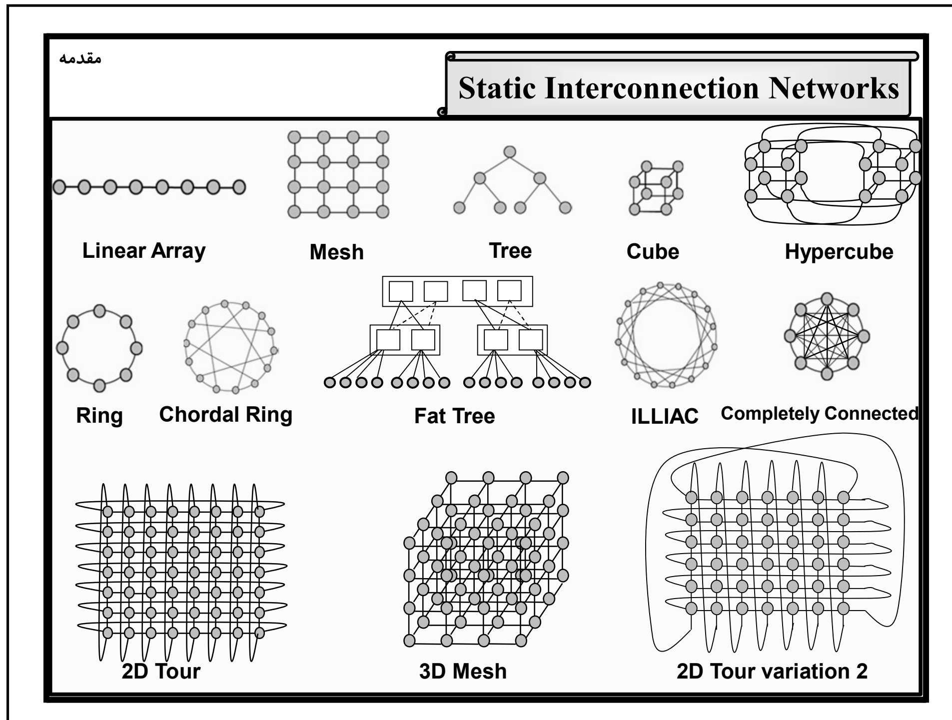
Interconnection Network SIMD Computers

- ❖ انواع Interconnection Networks (topology)
  - ➢ Dynamic
    - ▪ اتصالات بر اساس نیازها تنظیم می شوند.
  - ➢ Static
    - ▪ اتصالات ثابتی دارند.
- ❖ مدل های SIMD:
  - ➢ نسبت به مدل های MISD بسیار رایج تر هستند.
  - ➢ تحلیل، طراحی و پیاده سازی الگوریتم ها برای این مدل ها نسبتاً ساده است.
  - ➢ تنها زمانی قابل استفاده هستند که مسأله مورد نظر قابل تقسیم به تعدادی زیرمسأله یکسان باشد.
  - ➢ برای مسائلی که زیرمسائل آنها لزوماً یکسان نیستند، از مدل MIMD استفاده می کنیم.

مقدمه

Static Interconnection Networks





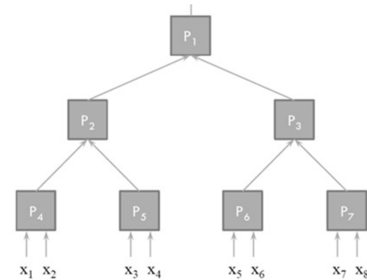
Parameters of some Static INs

- ❖ Degree: حداکثر تعداد اتصالات برای هر PE
- ❖ Diameter: حداکثر فاصله بین دو PE
- ❖ Bisection width: حداقل تعداد اتصالاتی که باید حذف شوند تا IN به دو قسمت تقسیم شود.

Topologies	Parameters		
	Degree	Diameter	Bisection width
Linear array	2	$N-1$	1
Ring	2	$\left\lfloor \frac{N}{2} \right\rfloor$	2
Completely Conn.	$N-1$	1	$\left(\frac{N}{2}\right)^2$
Binary tree	3	$2(log_2^N - 1)$	1
2D Mesh	4	$2(\sqrt{N}-1)$	$\sqrt{N}$
2D Torus	4	$\lfloor \sqrt{N} \rfloor$	$2\sqrt{N}$
Hypercube	log_2^N	log_2^N	$\frac{N}{2}$
2D mesh of Trees	3	$2log_2^N$	$\sqrt{N}$
Fat Tree	4	log_2^N	$\frac{\sqrt{N}}{log_2^N}$

محاسبه مجموع n عنصر با استفاده از tree

- ❖ محاسبه مجموع n عنصر در زمان $O(log\ N)$
- ❖ محاسبه مجموع m مجموعه از عناصر هر یک شامل n عنصر
  - ➢ روش ترتیبی: mn
  - ➢ روش موازی با استفاده از اتصال درختی: m(log n)
  - ➢ با استفاده از فرآیند pipelining: Log n + (m – 1)

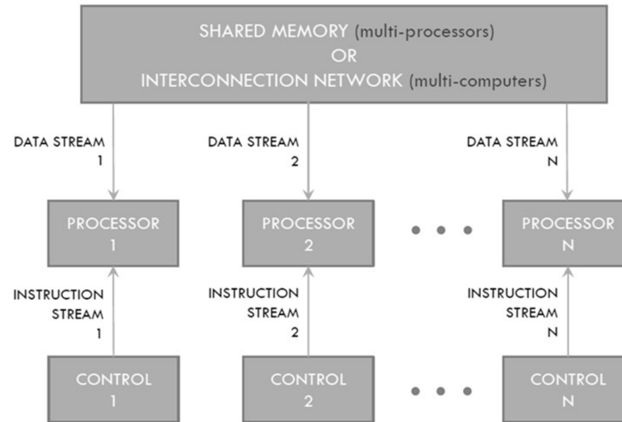


- ❖ درخت با N برگ 2N-1 پردازنده دارد.
- ❖ رابطه بین تعداد سطوح درخت و تعداد پردازنده ها عبارت است از:
  - ➢ $N=2^d-1$ (d سطح)

چنددستورالعمل، چند داده (MIMD)

- ❖ تمام پردازنده ها برنامه های مختلف را روی داده های مختلف اجرا می کنند تا زیرمسئله های یک مسئله را حل کنند.
- ❖ پردازنده ها به صورت asynchronous عمل می کنند.
- ❖ MIMD به دو دسته تقسیم می شوند:
- ❖ Multiprocessors (چندپردازنده ها) یا tightly coupled machines:
  - ➢ ماشین های پیوند قوی، کامپیوترهای MIMD با حافظه اشتراکی
  - ➢ انواع: EREW، CREW، ERCWو CRCW
- ❖ Multicomputer (چندکامپیوترها) یا loosely coupled machines:
  - ➢ Multicomputer ها سیستمهای توزیع شده هم نامیده می شوند.
  - ➢ ماشین های پیوند ضعیف، کامپیوترهای MIMD با IN
  - ➢ انواع بر اساس فاصله فیزیکی میان انواع پردازنده ها
    - ▪ فاصله نزدیک: چندکامپیوترها (مثلاً در یک اتاق)
    - ▪ فاصله دور: سیستم های توزیع شده (مثلاً در شهرهای مختلف)
      - ✺ تعداد انتقالات داده ای بسیار مهم تر از تعداد عملیات انجام شده

چنددستورالعمل، چند داده (MIMD)



- ❖ مثال: برنامه های کامپیوتری برای انجام بازی های راهبردی مانند شطرنج
- ❖ راه حل ترتیبی: جستجوی مسیرها در درخت بازی به صورت اول-عمق (depth-first)
- ❖ راه حل موازی: توزیع زیردرخت های ریشه در بین پردازنده ها

Systolic Arrays

- ❖ رایانه های Systolic کلاس جدیدی از معماری آرایه خط لوله (pipelined) هستند.
- ❖ سیستم های Systolic شامل مجموعه ای از PE (عناصر پردازش) می باشند.
- ❖ پردازنده ها سلول نامیده می شوند.
- ❖ هر سلول در یک شبکه مانند توپولوژی مش به تعداد کمی از نزدیکترین همسایه‌ها متصل است.
- ❖ هر سلول دنباله ای از عملیات را روی داده هایی که بین آنها جریان دارد انجام می دهد.
- ❖ به طور کلی عملیات در هر سلول یکسان خواهد بود، هر سلول یک عملیات یا تعداد کمی عملیات روی یک مورد داده را انجام می دهد و سپس آن را به همسایه خود منتقل می کند.

Systolic Arrays

❖ ویژگی های آرایه های Systolic (تپنده)

❖ آرایه Systolic یک شبکه محاسباتی است که دارای ویژگیهای زیر است:

- Synchrony,
- Modularity,
- Regularity,
- Spatial locality,
- Temporal locality,
- Pipelinability,
- Parallel computing.

Systolic Arrays

❖ ویژگی های آرایه های Systolic

❖ Synchrony (همگام سازی): داده ها به طور موزون محاسبه می شوند و از طریق شبکه منتقل می شوند.

❖ Modularity: آرایه (متناهی/نامتناهی) شامل واحدهای پردازش ماژولار است.

❖ Regularity (منظم بودن): واحدهای پردازشی ماژولار به طور یکنواخت به هم متصل هستند.

❖ Spatial locality : سلولها دارای ارتباطات interconnection محلی هستند.

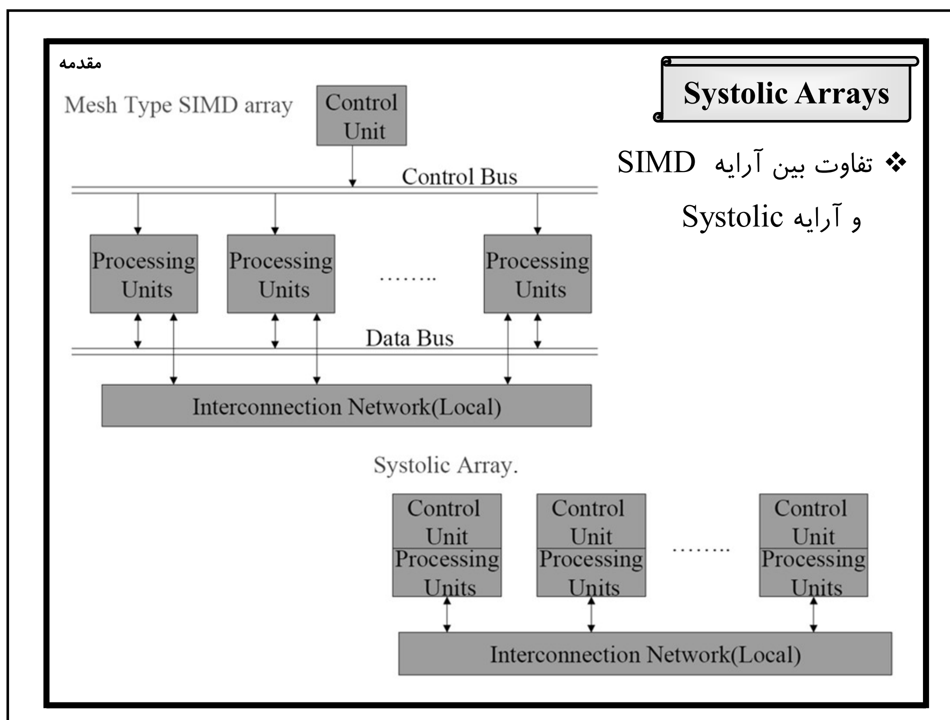
❖ Temporal locality: سلول ها سیگنال ها را از یک سلول به سلول دیگر منتقل می کنند که حداقل به یک واحد تاخیر زمان نیاز دارد.

❖ Pipelinability: آرایه می تواند به سرعت بالایی دست یابد.

مقدمه

Systolic Arrays

❖ تفاوت بین آرایه SIMD و آرایه Systolic



مقدمه

Systolic Arrays

❖ یک آرایه SIMD یک آرایه همزمان از PEها تحت نظارت یک واحد کنترل است و همه PEها دستورالعمل یکسانی را که واحد کنترل پخش می کند بر روی مجموعه داده های مختلف از جریان داده های مجزا اجرا می کنند.

❖ آرایه SIMD معمولاً داده ها را قبل از شروع محاسبه در حافظه محلی خود بارگذاری می کند.

❖ آرایه های Systolic معمولاً داده ها را از یک میزبان خارجی لوله (pipe) می کنند و نتایج را نیز به میزبان برمی گردانند.

❖ از Systolic Arrays می توان برای معماری پردازش با اهداف خاص استفاده کرد. به دلیل:

- طراحی ساده و منظم (Simple and Regular Design)
- همزمانی و ارتباطات (Concurrency and Communication)
- متعادل سازی محاسبه با ورودی/خروجی (Balancing Computation withI/O)

مقدمه

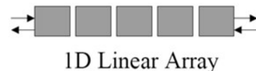
Systolic Arrays

- ❖ آرایه های Systolic دارای طراحی منظم و ساده هستند:
  - ✓ مقرون به صرفه است.
  - ✓ آرایه ماژولار است به این معنی که برای اهداف مختلف عملکرد قابل تنظیم است،
  - ✓ تعداد زیادی پردازنده با هم کار می کنند.
  - ✓ ارتباطات محلی در آرایه Systolic برای ارتباط سریعتر مفید است.
- ❖ آرایه Systolic به عنوان پردازنده آرایه ای متصل استفاده می شود.
  - ✓ داده ها را دریافت کرده و نتایج را از طریق رایانه میزبان متصل دریافت می کند.
  - ✓ بنابراین هدف عملکرد سیستم پردازنده آرایه نرخ محاسبه ای است که پهنای باند ورودی/خروجی را با میزبان متعادل می کند.
- ❖ با پهنای باند نسبتاً پایین دستگاههای ورودی/خروجی فعلی، برای دستیابی به سرعت محاسبه سریعتر، ضروری است که محاسبه های متعدد در هر دسترسی I/O انجام شود.

مقدمه

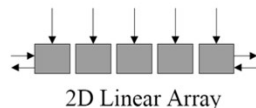
Systolic Arrays

- ❖ انواع Systolic Arrays
- ❖ آرایه های Systolic اولیه آرایه های خطی و یک بعدی (1D) یا دو بعدی I/O (2D) هستند.
- ❖ اخیراً، آرایه های Systolic به عنوان planar array با محیط ورودی/خروجی برای تغذیه داده ها از طریق مرز اجرا می شوند.
- ❖ آرایه خطی با ورودی/خروجی ۱ بعدی (Linear array with 1D I/O).
  - ➢ این پیکربندی برای ورودی/خروجی منفرد مناسب است.



1D Linear Array

- ❖ آرایه خطی با ورودی/خروجی ۲ بعدی.
  - ➢ اجازه می دهد تا کنترل بیشتری بر روی آرایه خطی داشته باشید.



2D Linear Array

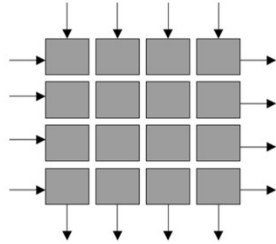
مقدمه

Systolic Arrays

❖ انواع Systolic Arrays

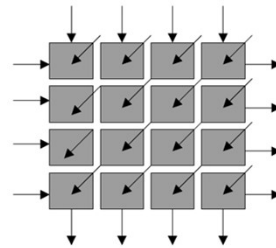
❖ .Planar array with perimeter I/O

➢ این پیکربندی فقط از طریق سلولهای مرزی خود اجازه ورود/خروج را می دهد.



❖ .Focal Plane array with 3D I/O

➢ این پیکربندی اجازه ورود/خروج به هر سلول Systolic را می دهد.



مقدمه

Systolic Arrays

❖ مزایای آرایه های Systolic عبارتند از:

- ❖ Regularity and modular design(Perfect for VLSI implementation).
- ❖ Local interconnections(Implements algorithms locality).
- ❖ High degree of pipelining.
- ❖ Highly synchronized multiprocessing.
- ❖ Simple I/O subsystem.
- ❖ Very efficient implementation for great variety of algorithms.
- ❖ High speed and Low cost.
- ❖ Elimination of global broadcasting and modular expansibility.

مقدمه

Systolic Arrays

❖ معایب اصلی آرایه های Systolic عبارتند از:

❖ Global synchronization limits due to signal delays.

❖ High bandwidth requirements both for periphery(RAM) and between PEs.

❖ Poor run-time fault tolerance due to lack of interconnection protocol.

❖ کاربردهای مختلف آرایه های Systolic :

❖ Matrix Inversion and Decomposition.
❖ Polynomial Evaluation.
❖ Convolution.
❖ Systolic arrays for matrix multiplication.
❖ Image Processing.
❖ Systolic lattice filters used for speech and seismic signal processing.
❖ Artificial neural network.

مقدمه

تحلیل الگوریتم ها

❖ معیارهای اندازه گیری زمان اجرای الگوریتم های ترتیبی:

- Sequential running time $t_s(n)$:
  - زمان اجرای الگوریتم از ابتدا تا انتها در بدترین حالت (با شمارش مراحل اصلی الگوریتم)
- Sequential space:
  - اندازه حافظه اصلی

❖ معیارهای اندازه گیری زمان اجرای الگوریتم های موازی:

- Parallel time
- Number of processors
- Space
- Cost
- Speedup
- Efficiency
- Scalability (feasibility)

مقدمه

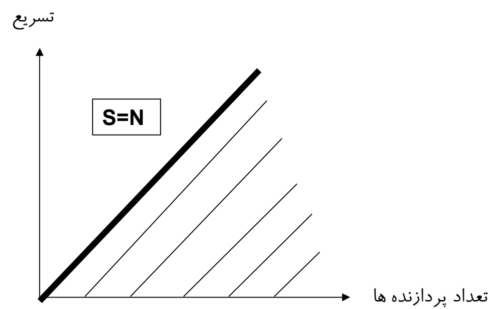
تحلیل الگوریتم ها

- ❖ $(t_p(n))$ Parallel time
  - زمان اجرای الگوریتم از شروع اولین پردازنده تا پایان کار آخرین پردازنده
- ❖ $(p(n))$ Number of processors
  - تعداد پردازنده هایی که برای اجرای الگوریتم استفاده می شود.
- ❖ Space
  - اندازه تمام حافظه هایی که برای اجرای الگوریتم استفاده می شود.
- ❖ Cost
  - زمان کل اجرای الگوریتم $C(n)=t_p(n)\times p(n)$
- ❖ (S_p) Speedup
  - زمان اجرای الگوریتم موازی با p پردازنده با زمان اجرای الگوریتم ترتیبی همان الگوریتم مقایسه می شود. $S_p=t_s/t_p$ $1\leq S_p\leq N$
- ❖ Efficiency
  - $E_p=S_p/p$
- ❖ Scalability (feasibility)
  - توانایی اضافه و حذف کردن PE، فاکتور مهم در تشخیص کارا بودن الگوریتم در داده های بزرگ. با اضافه شدن پردازنده های جدید، پردازنده ها با همان کارایی قبل کار کنند.

مقدمه

تحلیل الگوریتم ها

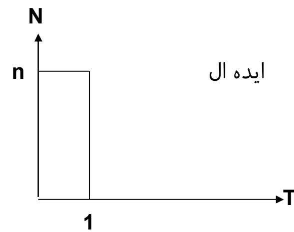
- ❖ در حالت ایده ال توقع داریم تسریع با افزایش تعداد پردازنده ها به n، برابر با n شود.
- ❖ پس اگر N تعدادپردازنده ها و S=N باشد داریم: $E = N/N*100\%= 100\%$
- ❖ تسریع همیشه زیر خط S=N قرار دارد.

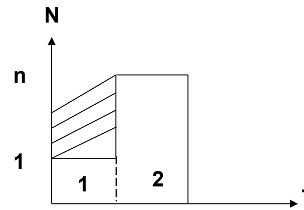


مقدمه

قانون Amdahl

- ❖ هر مساله شامل دو بخش است بخشی که پردازش سریال دارد (یک پردازنده) و بخشی که پردازش موازی دارد (n پردازنده)





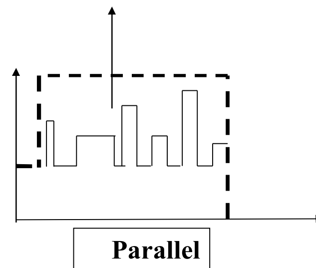
۱: پردازش سریال (ایده آل این است که کوچکتر شود)

۲: پردازش موازی

مقدمه

قانون Amdahl

- ❖ همواره نمی توان به تسریع ایده آل رسید زیرا بخشی از پردازش به صورت sequential است. هر الگوریتمی با یک پردازنده شروع به کار می کند.
- ❖ در همه فضاهای خالی کارایی را از دست می دهیم.



Parallel

- ❖ بر اساس این قانون، ایده آل زمانی است که S=N. اما به دلیل شروع کار با یک پردازنده و ...حداکثر کارایی کمتر از خط S=N است.

قانون Amdahl

❖ مدل ریاضی برای نشان دادن کارایی در محاسبات موازی

❖ $$R(f) = \frac{1}{\frac{f}{R_H} + \frac{1-f}{R_L}}$$

❖ R_H :High Execution Rate

❖ R_L :Low Execution Rate

❖ f: نسبت پردازش موازی برای برنامه مورد نظر

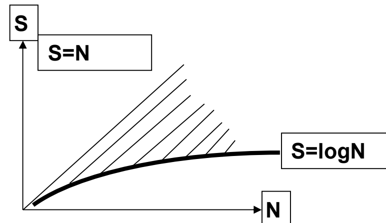
❖ 1-f: نسبت پردازش سریال برای برنامه مورد نظر

❖ باید یک حد پایین نیز برای الگوریتم های موازی مشخص کرد.

قانون Minsky

❖ حداقل تسریع که توقع داریم تسریع نمایی است.

❖ پس باید الگوریتم موازی در فضای هاشور خورده زیر باشد:



❖ به دلیل وابستگی ها (dependency) به تسریع ایده آل نمی رسیم.

مقدمه

قانون Worlton

- بحث در ارتباط با محدودیتهایی که در پردازش موازی صورت می گیرد، است. آنچه باعث می شود تسریع در پردازنده های موازی مطرح باشد.
- هر مساله شامل N، task است که زمان متوسط اجرای هر task، t است.
- t_s زمان سنکرونیزاسیون بین task ها است.
- t_o: زمان overhead
- زمانی که برای یکسری task سربار قرار دهید. بخشی از task1 را به task2 ببریم تا task2 به طور مستقل کار انجام دهد. Overhead دارد اما استقلال ایجاد شده است.
- $T_{seq} = N.t$
- $T_{par} = t_s + [N/P](t+t_o)$
- $(t+t_o)$: زمان اجرای هر task
- N/P: تعداد Stepها
- P: تعداد پردازنده ها

مقدمه

قانون Worlton

- تسریع برابر است با:

$$S = \frac{N.t}{t_s + \left[\frac{N}{P}\right](t + t_o)}$$

- طرفین را بر N.t تقسیم می کنیم:

$$S = \frac{1}{\frac{t_s}{N.t} + \left(\frac{1}{N}\right)\left[\frac{N}{P}\right]\left(1 + \frac{t_o}{t}\right)}$$

$$S = \frac{1}{\frac{t_s}{N.t} + \left[\frac{1}{P}\right]\left(1 + \frac{t_o}{t}\right)}$$

N↓ ⇨t↑

N ↑ ⇨t ↓

قانون Worlton

❖ برای تسریع حداکثر:

۱- کاهش زمان t_s،t_o ۲- افزایش t ۳- افزایش p ۴- افزایش N

$$S = \frac{1}{\frac{t_s}{N.t} + \left[\frac{1}{P}\right]\left(1 + \frac{t_o}{t}\right)}$$

❖ افزایش تعداد پردازندها به دلیل اینکه کارایی را کم می کند مناسب نیست:

❖ $P \to \infty$: $E=S / P \to 0$

❖ با افزایش N ($N\to\infty$) در فرمول اول داریم:

❖ $N \to \infty$: $S=\infty / \infty$

➢ $P \to \infty$, N:fix : $S = \frac{1}{\frac{t_s}{N.t}} = \frac{N.t}{t_s}$

➢ $N \to \infty$, P:fix : $S = \frac{P}{\left(1 + \frac{t_o}{t}\right)} = \frac{P.t}{t + t_o}$

انواع dependency

❖ داده (Data)

- Data Flow Dependency (ماهیت برنامه)
- Anti Data Dependency (برنامه نویس)
- Output Data Dependency (برنامه نویس)

❖ کنترل (Control)

- if … then …. else

❖ منبع (Resource)

- اگر دو instruction که همزمان کار می کنند بخواهند از یک منبع استفاده کنند...

✓ وابستگی ها: ۱) یا در ماهیت خود برنامه وجود دارند

۲) یا برنامه نویس ایجاد می کند

✓ وابستگی دوم قابل حذف کردن است.

✓ یکی از وظایف کامپایلر حذف وابستگی هایی است که برنامه نویس ایجاد می کند و قبل از مرحله exe انجام می شود و سبب کارایی بالای برنامه می شود.

Dependency

Data Flow Dependency

s_i: O……=…….O

s_j: O……=……...O

- وابستگی است که از سمت چپ به سمت راست instruction است.
- به این وابستگی write before read (WBR) یا read after write (RAW) گفته می شود.

- مثال ۱:

s_i: A=B+C
s_j: K=A+C

- مثال ۲:

s_1: A=B+C
s_2: E=A-D

Dependency

Anti Data Dependency

s_i: O……=…….O

s_j: O……=……...O

- به این وابستگی read before write (RBW) گفته می شود.

- مثال ۱:

s_i: B=C+A
s_j: A=K+J

- مثال ۲:

s_1: B=A*D
s_2: A=F-5

Dependency

Output Data Dependency

s_i: O……=…….O

↓

s_j: O……=……....O

- ❖ وابستگی است که از سمت چپ به سمت چپ instruction است.
- ❖ به این وابستگی write before write(WBW) یا write after write (WAW) گفته می شود.

- ❖ مثال ۱:

s_i: A=B+C
s_j: A=K+J

- ❖ مثال ۲:

s_1: A=B+C
s_2: A=D+E

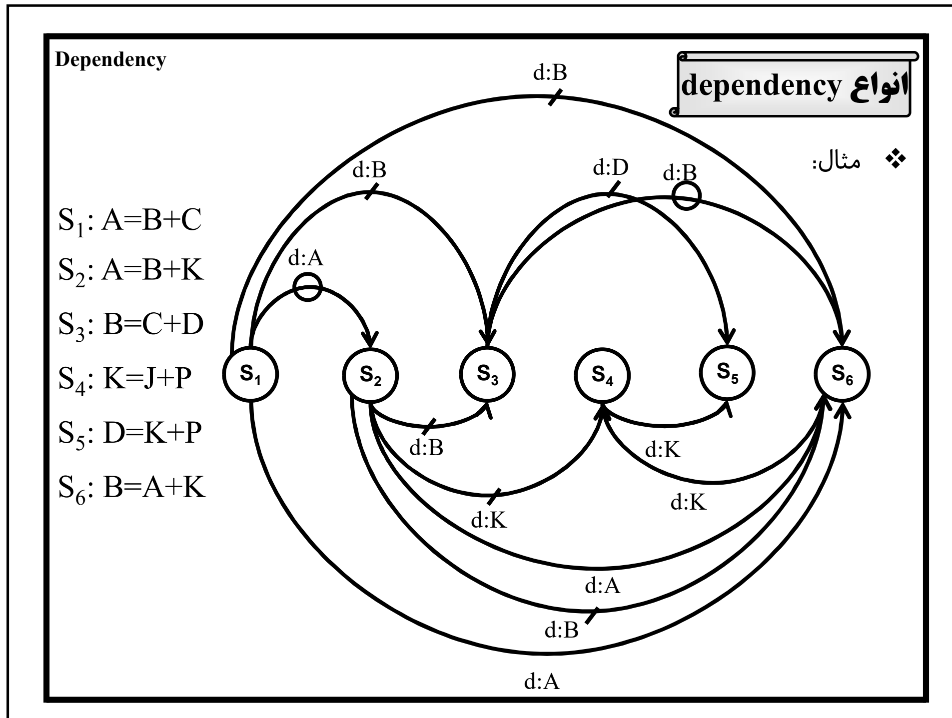
Dependency

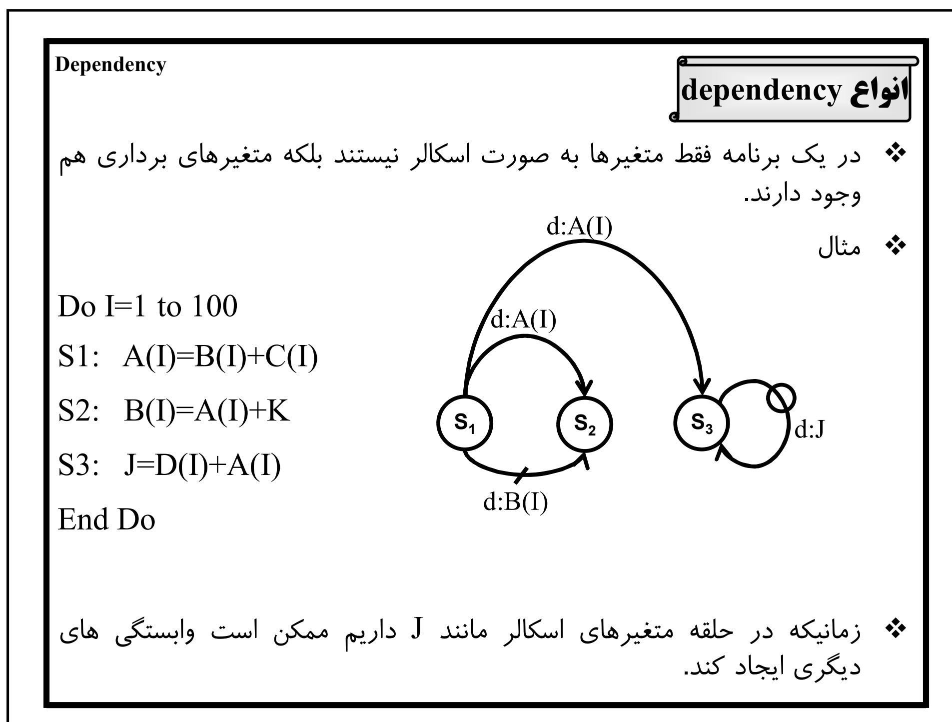
انواع dependency

- ❖ برای نشان دادن وابستگی ها از گراف استفاده می شود.
- ❖ اگر سوی همه گره ها به یک سمت (هم جهت) باشد و حلقه نداشته باشیم گراف آبشاری است که در این حالت می توان با performance مناسبی پردازش را انجام داد.
- ❖ بدترین نوع حالتی است که در گراف وابستگی حلقه داشته باشیم.
- ❖ وابستگی ها با → نشان داده می شود.

- ❖ write ⟶ read
- ❖ read ⟶ (arrow with slash) write
- ❖ write ⟶ (arrow with circle) write

- ❖ پس از اینکه وابستگی ها نشان داده شد به صورت کد در می آید و در یک dependency Table نشان داده می شود.


Dependency
انواع dependency
❖ مثال:
S1: A=B+C
S2: A=B+K
S3: B=C+D
S4: K=J+P
S5: D=K+P
S6: B=A+K
d:B
d:B
d:D
d:B
d:A
d:B
d:K
d:K
d:K
d:A
d:B
d:A


Dependency
انواع dependency
❖ در یک برنامه فقط متغیرها به صورت اسکالر نیستند بلکه متغیرهای برداری هم وجود دارند.
❖ مثال
Do I=1 to 100
S1: A(I)=B(I)+C(I)
S2: B(I)=A(I)+K
S3: J=D(I)+A(I)
End Do
d:A(I)
d:A(I)
d:J
d:B(I)
❖ زمانیکه در حلقه متغیرهای اسکالر مانند J داریم ممکن است وابستگی های دیگری ایجاد کند.

Dependency

انواع dependency

❖ مثال:

```
Do I=1 to 100
S1:   A(I)=B(I)+C(I)
S2:   B(I)=A(I)+K
S3:   K=D(I)+A(I)
End Do
```


❖ بین S2 وS3 حلقه ایجاد شده است و این حلقه باعث کاهش موازی سازی می شود.

Dependency

انواع dependency

❖ مثال:

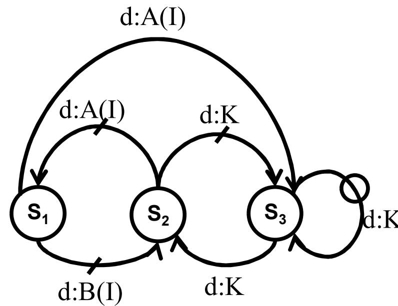
```
Do I=1 to 100
S1:   A(I)=B(I)+C(I)
S2:   B(I)=A(I-1)+K
S3:   J=P(I)+A(I)
End Do
```

Dependency

انواع dependency

❖ مثال:

```
Do I=1 to 100
S1:   A(I)=B(I)+C(I)
S2:   B(I)=A(I+1)+K
S3:   K=P(I)+A(I)
End Do
```



❖ کامپایلر K را به متغیر برداری تبدیل می کند.

❖ تنوع بیشتری در تعریف متغیرها داشته باشید تا وابستگی کمتر شود.

Dependency

انواع dependency

❖ حلقه ها به دو شکل مطرح می شوند:

- Do All: index ها یکسان است.
- Do Across: index ها متفاوت است.

❖ متغیر های اسکالر باعث ایجاد وابستگی بین تکرارهای مختلف می شود.

❖ در حلقه های Do All متغیر های برداری باعث ایجاد وابستگی نمی شوند.

❖ در حلقه های Do Across متغیرهای برداری نیز باعث ایجاد وابستگی می شوند.

❖ وابستگی data flow به ذات برنامه مربوط است.

❖ وابستگی های Anti data و Output data معمولاً بخاطر رعایت نشدن برنامه نویسی موازی رخ می دهد.

Dependency

حذف dependency

- سه روش برای حذف وابستگی ها وجود دارد:
  - Variable Renaming
  - Scalar Expansion
  - Node Splitting
- Variable Renaming
- مثال:

S1: A=B*C

S2: D=A+1

S3: A=A*D

Dependency

حذف dependency

- در قانون S3 روی A، Renaming انجام می شود.

S3:AA=A*D

S1: A=B*C

S2: D=A+1

S3: AA=A*D

Dependency

حذف dependency

- Scalar Expansion
- هدفش حذف وابستگی های ناشی از متغیر های اسکالر می باشد پس یک متغیر اسکالر را به یک متغیر برداری تغییر می دهد.
- مثال:

```
FOR I=1 to 100 do
S1:  b=B(I)-2
S2:  C=C'(I)-B(I)
S3:  A(I)=b+C
END
```


Dependency

حذف dependency

- تغییرات مقابل را اعمال و برنامه را بازنویسی میکنیم:

b→b'(I)

C → C(I)

- مثال:

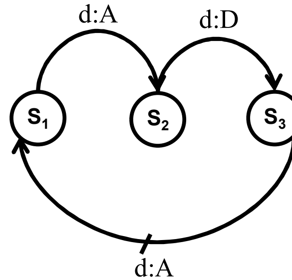
```
FOR I=1 to 100 do
S1:  b'(I)=B(I)-2
S2:  C(I)=C'(I)-B(I)
S3:  A(I)=b'(I)+C(I)
END
```

Dependency

حذف dependency

- ❖ Node Splitting
- ❖ وابستگی بین متغیرهای برداری در حلقه های do across را حذف می کند. برای این منظور متغیر برداری را با یک متغیر برداری دیگر rename می کند تا اندیس های انها یکسان شود و تبدیل به یک حلقه do all گردد.
- ❖ مثال:

```
FOR I=1 to 100 do
S1: A(I)=B(I)-C(I)
S2: D(I)=A(I)+2
S3: F(I)=D(I)+A(I+1)
END
```

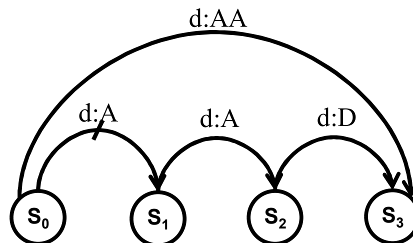


Dependency

حذف dependency

- ❖ Node Splitting

```
FOR I=1 to 100 do
S0: AA(I)=A(I+1)
S1: A(I)=B(I)-C(I)
S2: D(I)=A(I)+2
S3: F(I)=D(I)+AA(I)
END
```

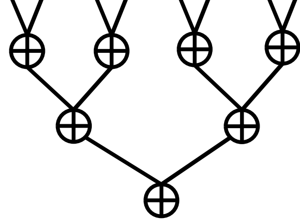


Parallel in statement

توازی در statement

- ❖ توازی می تواند بین statement ها یا درون statement باشد.
- ❖ توازی در درون statement

❖ $S=a_1+a_2+a_3+a_4+a_5+a_6+a_7+a_8$ O(n)



- ❖ کامپایلر با در نظر گرفتن قوانینی مانند قوانین Brent می تواند حداکثر توازی سازی را انجام دهد.
- ❖ همچنین قوانینی مانند recursive doubling برای تبدیل درخت محاسبه نامتقارن به متقارن جهت ایجاد حداکثر توازی سازی وجود دارد. اکثر مشکلات در الگوریتم موازی به وابستگی داده بر میگردد.

Parallel in statement

توازی درون statement

- ❖ در سال ۱۹۷۰ Brent سه قاعده مطرح کرد که این قوانین مربوط به درون جمله است و هدف این قواعد کاهش عمق درخت محاسبه است.
- ❖ این سه قانون عبارتند از:
- ❖ شرکت پذیری (Associative Rule)
- ❖ جابجایی (Commutative Rule)
- ❖ توزیع پذیری (Distributive Rule)
- ❖ در پردازش موازی داخل جمله می توان این توقع را داشت که O(N) را به O(logN) ببریم.

Parallel in statement

توازی درون statement

❖ شرکت پذیری (Associative Rule)

❖ (((a+b)×c)×d) => (a+b)×(c×d)

❖ جابجایی (Commutative Rule)

❖ a×(b+c)×d => (a×d)×(b+c)

❖ توزیع پذیری (Distributive Rule)

❖ a×b×c+a×b => (a×b) ×(c+1)

Parallel in statement

توازی مابین statementها

❖ بحث توازی مابین جملات نیز مطرح است.

❖ S1: x=a+bcd

❖ S2: y=ex+f

❖ S3: z=my+x

a b c d e f m

S1

S2

S3

Steps=7

Parallel in statement

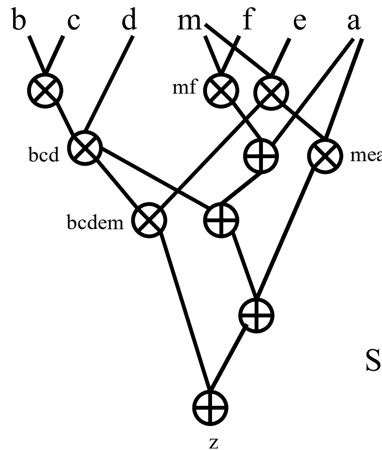
توازی مابین statementها

❖ بحث توازی مابین جملات نیز مطرح است. z=a+mf+bcd+mea+bcdem

❖ S1: x=a+bcd

❖ S2: y=ex+f

❖ S3: z=my+x



Steps=5

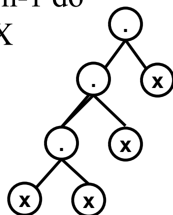
Parallel in statement

Recursive Doubling

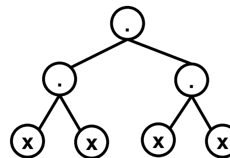
❖ کامپایلرها برای اینکه بتواند با حداقل تعداد عملیات کار را انجام دهند و وابستگی بین جملات حداقل شود اینکار را انجام میدهد.

❖ هدف این است که پردازشهایی که به صورت linear است به logarithm تبدیل شود.

```
begin
    S=X
    for i=1 to n-1 do
        S=S*X
end
```




```
begin
    for i=1 to k do        k=logn
        S=S*S
end
```



Broadcasting a Datum

انتشار داده

procedure BROADCAST(D, N, A)

Step 1: Processor P_1,

(i) reads the value in D

(ii) stores it in its own memory

(iii) writes it in A(1)

Step 2: for i = 0 to (Log N – 1) do

for $j = 2^i + 1$ to 2^{i+1} do in parallel

Processor P_j

(i) reads the value in $A(j - 2^i)$

(ii) stores it in its own memory

(iii) writes it in A(j)

end for

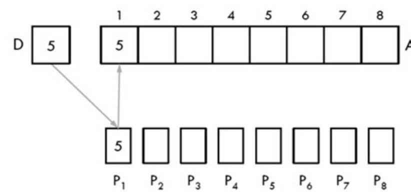
end for

- t(n) = O(log N)

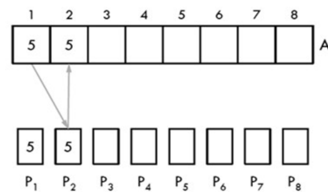
Broadcasting a Datum

انتشار داده

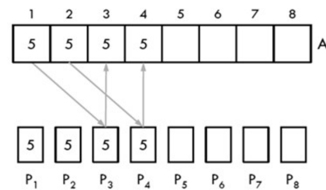
STEP 1



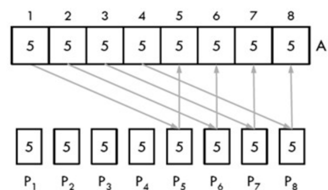
STEP 2 (i = 0)



STEP 2 (i = 1)



STEP 2 (i = 2)



Computing All Sums

محاسبه مجموع

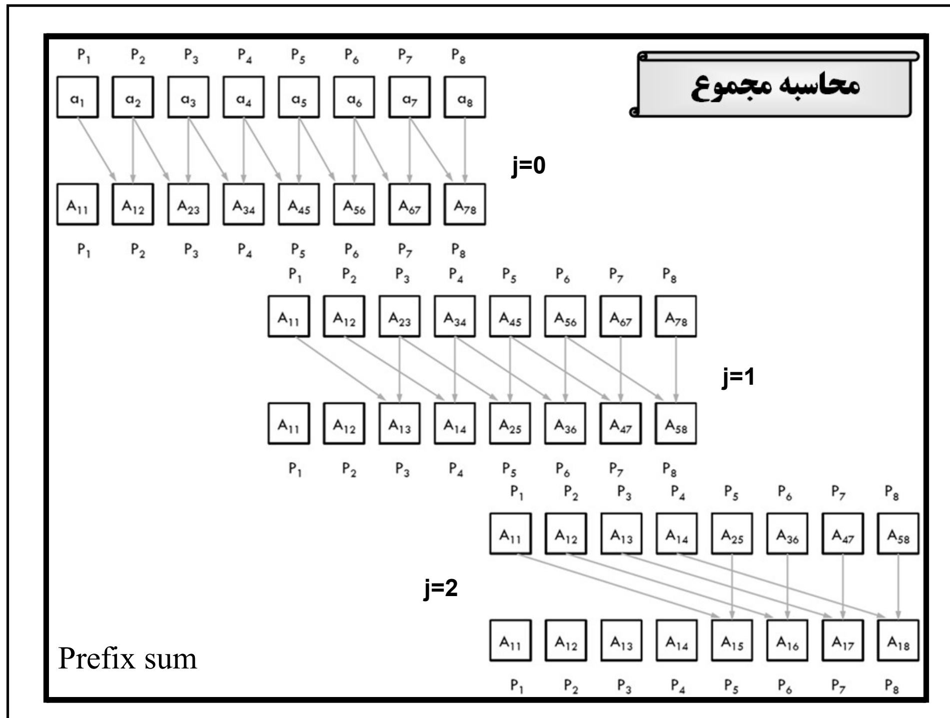
- فرض می کنیم پردازنده P_i در حافظه محلی خود مقدار عددی a_i را نگهداری می کند (به ازای $1 \le i \le N$)
- اکنون می خواهیم به ازای هر پردازنده مانند P_i مجموع زیر را محاسبه کنیم:
  - $a_1 + a_2 + \ldots + a_i$
- هدف: مشغول نگاه داشتن پردازنده ها تا حد ممکن و استفاده از خاصیت شرکت پذیری در عمل جمع

Computing All Sums

محاسبه مجموع

```
procedure ALLSUMS(a1, a2, … , an)
for j = 0 to Log N – 1 do
        for i = 2^j + 1 to N do in parallel
                Processor P_i
                  (i) obtains a_{i-2^j} from P_{i-2^j}, through shared memory
                  (ii) replaces a_i with a_{i-2^j}+a_i
        end for
end for
```

- $t(n) = O(\log N)$



Prefix Sum on Tree

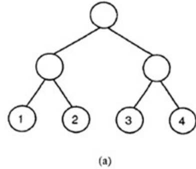
محاسبه پیشوند مجموع

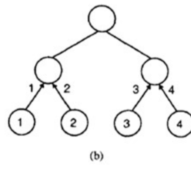
- Root Processor
  - (1) if an input is received from the left child then send it to the right child
    end if.
  - (2) if an input is received from the right child then discard it
    end if.
- Intermediate Processor
  - (1) if an input is received from the left and right children then
    - (i) send the sum of the two inputs to the parent
    - (ii) send the left input to the right child

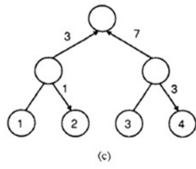
    end if.
  - (2) if an input is received from the parent then
    send it to the left and right children
    end if.
- Leaf Processor P_i
  - (1) $s_i \leftarrow x_i$.
  - (2) send the value of xi to the parent.
  - (3) if an input is received from the parent then add it to s_i
    end if.

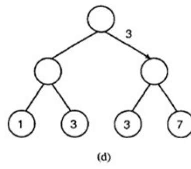
Prefix Sum on Tree

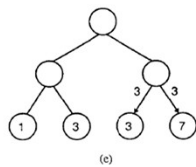
محاسبه پیشوند مجموع

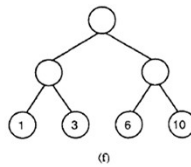












- inputs: x_i
- processors: P_i
- prefix sums of x_i : s_i

- تحلیل:
  - $t(n) = O(2 \log n)$
  - $t(n) = O(\log n)$
  - $p(n) = O(2n-1)= O(n)$
  - $c(n) = O(n \log n)$

Prefix Sum on Mesh

محاسبه پیشوند مجموع

- first step

  with all rows operating in parallel, the prefix sums for the elements in each row are computed sequentially:

  Each processor adds to its contents the contents of its left neighbour.

- second step

  the prefix sums of the contents in the rightmost column are computed.

- Finally

  again with all rows operating in parallel, the contents of the rightmost processor in row k - 1 are added to those of all the processors in row k (except the rightmost).

Prefix Sum on Mesh

محاسبه پیشوند مجموع

x_0	x_1	x_2	x_3
x_4	x_5	x_6	x_7
x_8	x_9	x_{10}	x_{11}
x_{12}	x_{13}	x_{14}	x_{15}

(a) INITIALLY

x_0	A_{01}	A_{02}	A_{03}
x_4	A_{45}	A_{46}	A_{47}
x_8	A_{89}	$A_{8,10}$	$A_{8,11}$
x_{12}	$A_{12,13}$	$A_{12,14}$	$A_{12,15}$

(b) AFTER STEP 1

x_0	A_{01}	A_{02}	A_{03}
x_4	A_{45}	A_{46}	A_{07}
x_8	A_{89}	$A_{8,10}$	$A_{0,11}$
x_{12}	$A_{12,13}$	$A_{12,14}$	$A_{0,15}$

(c) AFTER STEP 2

x_0	A_{01}	A_{02}	A_{03}
A_{04}	A_{05}	A_{06}	A_{07}
A_{08}	A_{09}	$A_{0,10}$	$A_{0,11}$
$A_{0,12}$	$A_{0,13}$	$A_{0,14}$	$A_{0,15}$

(d) AFTER STEP 3

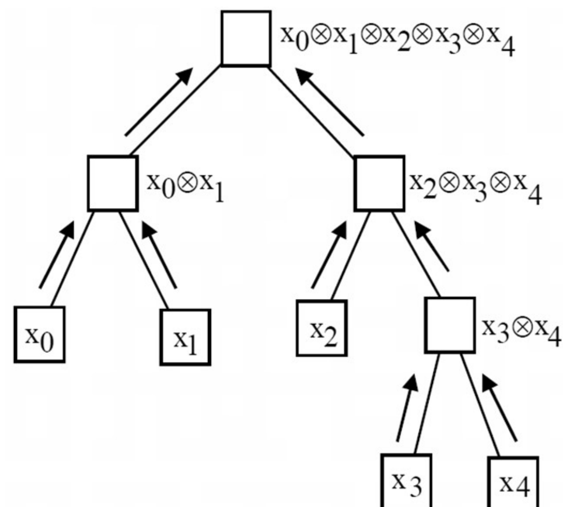
- rows: k
- columns: j
- inputs: x_i
- processors: P_i
- prefix sums of x_i : s_i

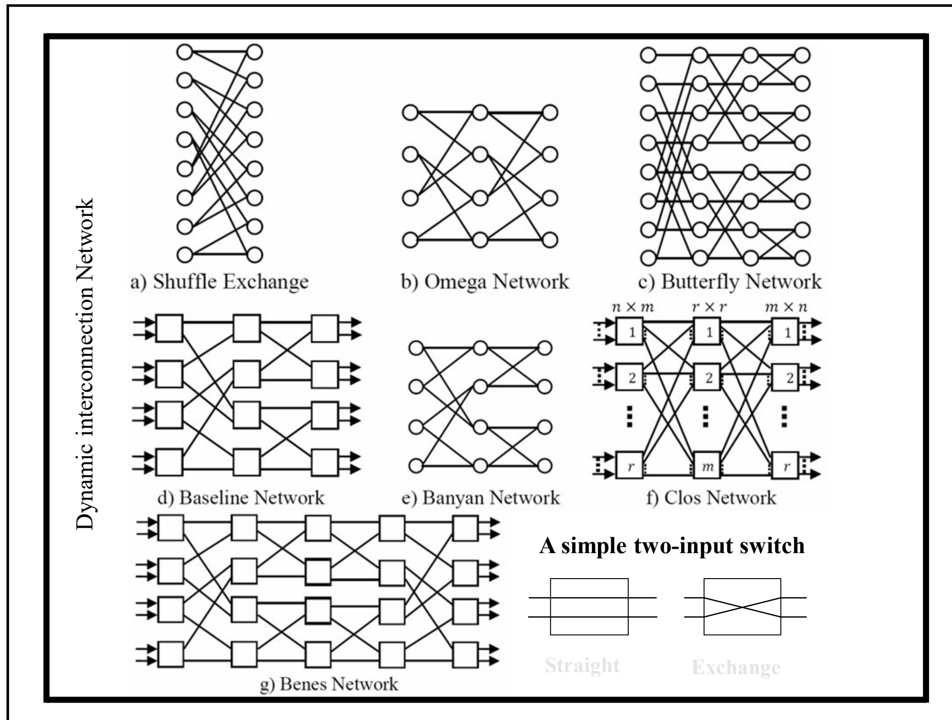
- تحلیل:
  - $t(n) = O(n^{1/2})$
  - $p(n) = O(n)$
  - $c(n) = O(n^{3/2})$

Computing All Sums (Tree)

محاسبه مجموع

- $t(n) = O(\log N)$





Computing All Sums (Shuffle-Exchange)

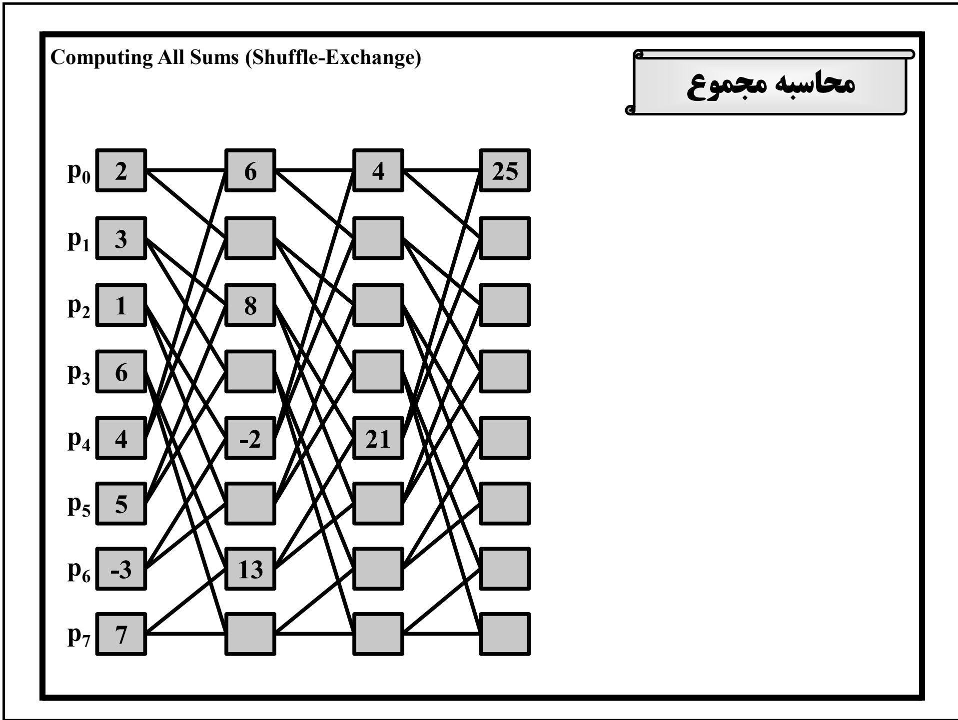
محاسبه مجموع

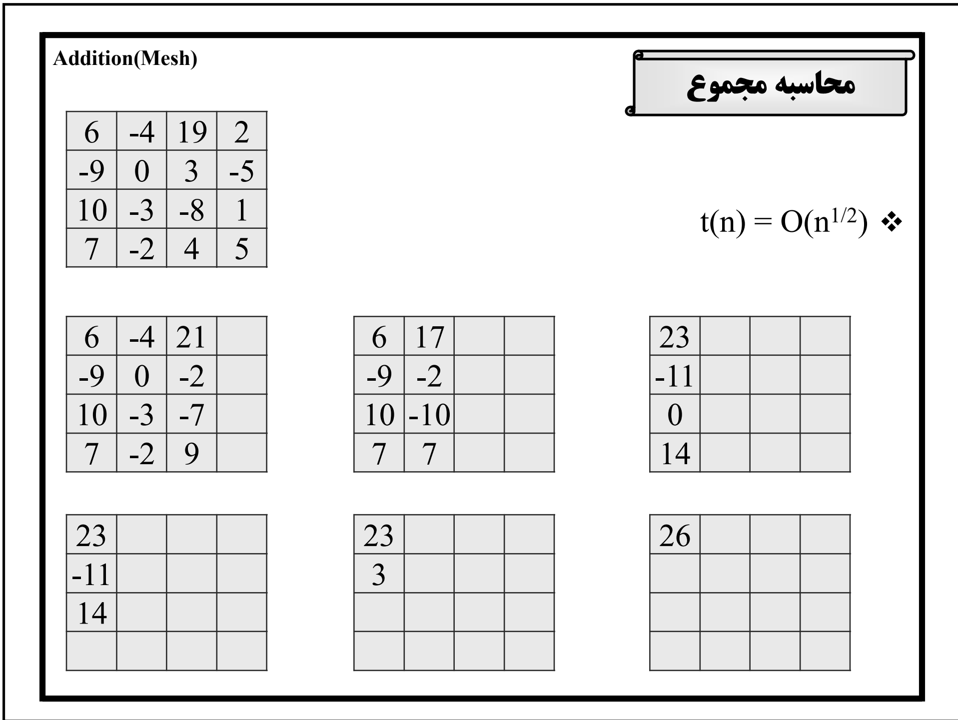
❖ محاسبه مجموع 2^n داده $(a_0,a_1,a_2,\ldots,a_{k-1})$

❖ هر پردازنده دو متغیر محلی دارد (s_i: که مقدار a_i را دارد و t_i: که مقدار اولیه صفر را دارد برای $0 \leq i \leq k-1$)

❖ هر پردازنده، داده پردازنده همسایه را در متغیر t_i قرار می دهد.

```
SHUFFLE EXCHANGE SIMD
for i = 1 to log k do
        for all s_j where 0 ≤ j ≤ k-1 do in parallel
                Shuffle ( s_j )
                t_j = s_j
                Exchange ( t_j )
                s_j = s_j + t_j
        end For
end For
```





M. Damrudi

Matrix operation

Matrix operation

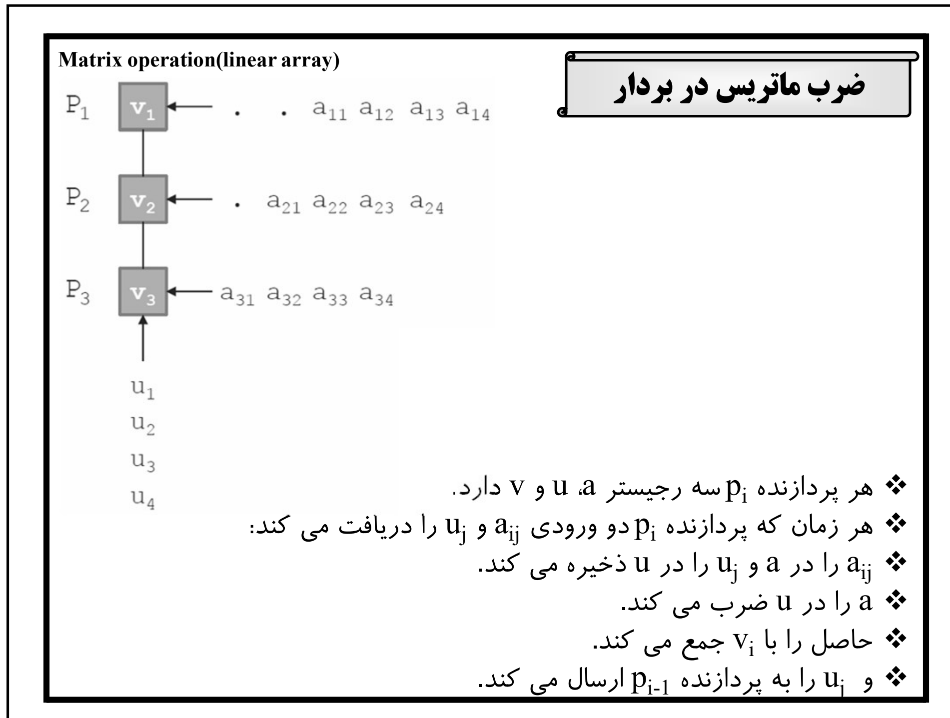
- عملیات مربوط به ماتریس:
  - محاسبه حاصل ضرب یک ماتریس در یک بردار
  - محاسبه حاصل ضرب دو ماتریس
  - محاسبه ترانهاده یک ماتریس
- برخی از کاربردها:
  - Convolution
  - حل یک دستگاه معادلات
  - بازنمایی و عملیات گراف ها

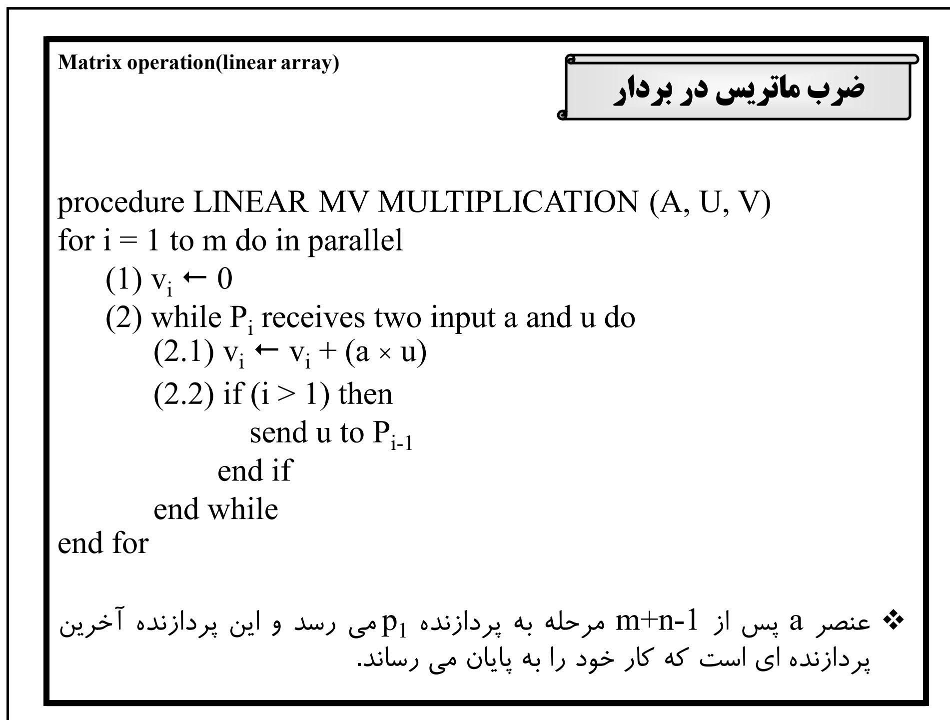
Matrix operation

ضرب ماتریس در بردار

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}_{m \times n} \times \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix}_{n \times 1} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}_{m \times 1}$$

- این مسأله یک حالت خاص از مسأله ضرب ماتریس در ماتریس است. به منظور نمایش نحوه استفاده از دو IN (tree, linear array)، این مسأله را به صورت جداگانه بررسی می شود.
  - Vector to matrix Multiplication
  - Tree Multiplication


Matrix operation(linear array)
ضرب ماتریس در بردار
P1
P2
P3
v1
v2
v3
a11 a12 a13 a14
a21 a22 a23 a24
a31 a32 a33 a34
u1
u2
u3
u4
❖ هر پردازنده p_i سه رجیستر a، u و v دارد.
❖ هر زمان که پردازنده p_i دو ورودی a_{ij} و u_j را دریافت می کند:
❖ a_{ij} را در a و u_j را در u ذخیره می کند.
❖ a را در u ضرب می کند.
❖ حاصل را با v_i جمع می کند.
❖ و u_j را به پردازنده p_{i-1} ارسال می کند.


Matrix operation(linear array)
ضرب ماتریس در بردار
procedure LINEAR MV MULTIPLICATION (A, U, V)
for i = 1 to m do in parallel
(1) $v_i \leftarrow 0$
(2) while P_i receives two input a and u do
(2.1) $v_i \leftarrow v_i + (a \times u)$
(2.2) if (i > 1) then
send u to P_{i-1}
end if
end while
end for
❖ عنصر a پس از m+n-1 مرحله به پردازنده p_1 می رسد و این پردازنده آخرین پردازنده ای است که کار خود را به پایان می رساند.

Matrix operation(linear array)

ضرب ماتریس در بردار

❖ تحلیل:

- $t(n) = O(m + n - 1)$

- اگر فرض شود $m \leq n$ آنگاه:

- $t(n) = O(n)$
- $p(n) = O(m) = O(n)$
- $c(n) = O(n^2)$

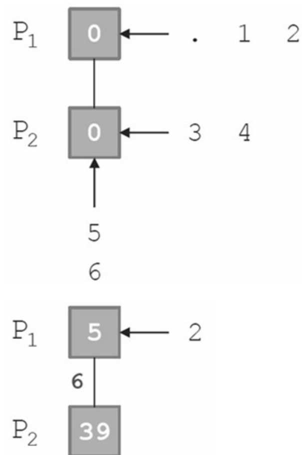
❖ مثال:

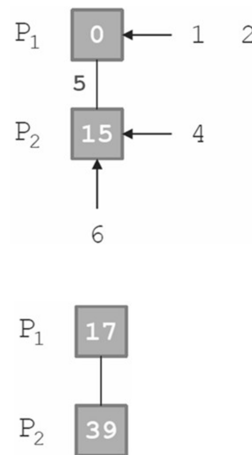
$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 17 \\ 39 \end{pmatrix}$$

Matrix operation(linear array)

ضرب ماتریس در بردار

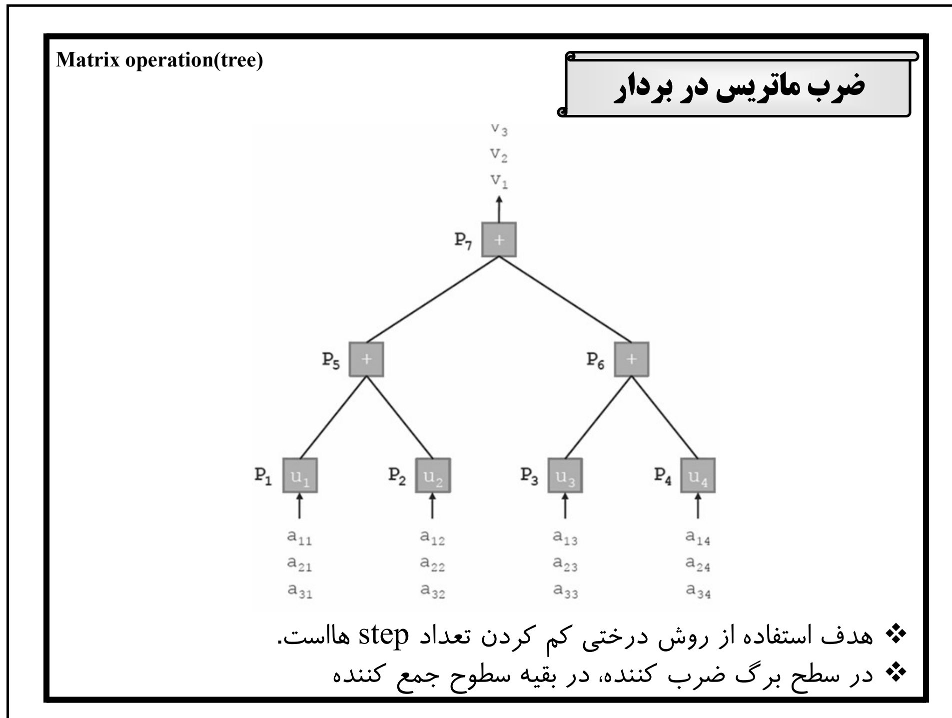
$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 17 \\ 39 \end{pmatrix}$$





Matrix operation(tree)

ضرب ماتریس در بردار



- هدف استفاده از روش درختی کم کردن تعداد step هاست.
- در سطح برگ ضرب کننده، در بقیه سطوح جمع کننده

Matrix operation(tree)

ضرب ماتریس در بردار

```
procedure TREEMV MULTIPLICATION (A, U, V)
do step 1 and 2 in parallel
      (1) for i = 1 to n do in parallel
                  for j = 1 to m do
                     (1.1) compute u_i × a_ji
                     (1.2) send result to parent
                  end for
           end for
      (2) for i = n + 1 to 2n – 1 do in parallel
                  while P_i receives two input do
                        (2.1) compute the sum of two inputs
                       (2.2) if (i < 2n-1) then
                                       send the result to parent
                               else
                                       produce the result as output
                              end if
                  end while
         end for
```

Matrix operation(tree)

ضرب ماتریس در بردار

❖ تحلیل: (تعداد سطرها m)

❖ اولین خروجی پس از logn مرحله و m-1 خروجی، هر کدام پس از یک مرحله تولید می شوند. بنابراین:

- $t(n) = O(m + \log n - 1)$
- $p(n) = 2n - 1 = O(n)$
- $c(n)=O(mn + n\log n - n)$
- $c(n)=O(n^2 + n\log n - n)$ if $m=n$

➢ اگر فرض شود $m \leq n$ آنگاه:

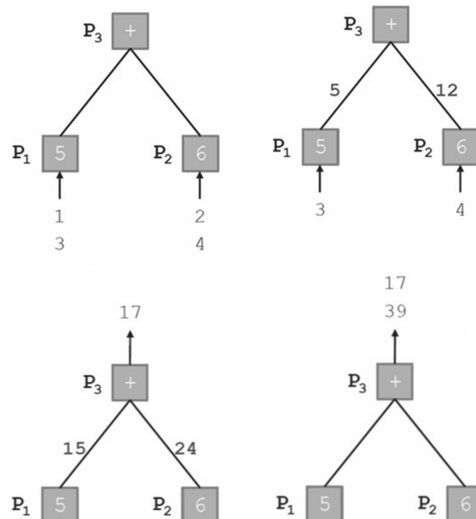
- $t(n) = O(n)$
- $p(n) = 2n - 1 = O(n)$
- $c(n) = O(n^2)$

Matrix operation(tree)

ضرب ماتریس در بردار

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 17 \\ 39 \end{pmatrix}$$

❖ مثال:



Matrix operation(tree)

ضرب ماتریس در بردار

- الگوریتم موازی ضرب ماتریس در بردار برای حل مسائل Convolution استفاده می شود که یکی از کاربردهای این ضرب است.
- دنباله ای از ثابت های $\{w_1,w_2,\ldots,w_n\}$ و ورودی های $\{x_1,x_2,\ldots,x_n\}$ را در نظر بگیرید. دنباله خروجی $\{y_1,y_2,\ldots,y_{2n-1}\}$ محاسبه می شود به گونه ای که

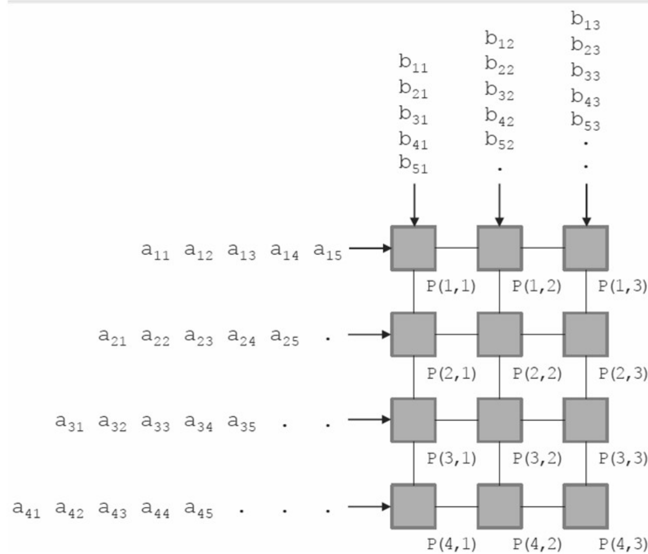
➢ $$y_i = \sum_{j=1}^{n} x_{i-j+1} \times w_j, \qquad 1 \le i \le 2n-1$$

- این محاسبه که می تواند به صورت ضرب ماتریس در بردار فرموله شود، به عنوان کانولوشن شناخته می شود و در پردازش سیگنال دیجیتال مهم است. برای n=3 داریم:

$$\begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ 0 & x_3 & x_2 \\ 0 & 0 & x_3 \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

Matrix operation(mesh)

ضرب ماتریس در ماتریس



Matrix operation(mesh)

ضرب ماتریس در ماتریس

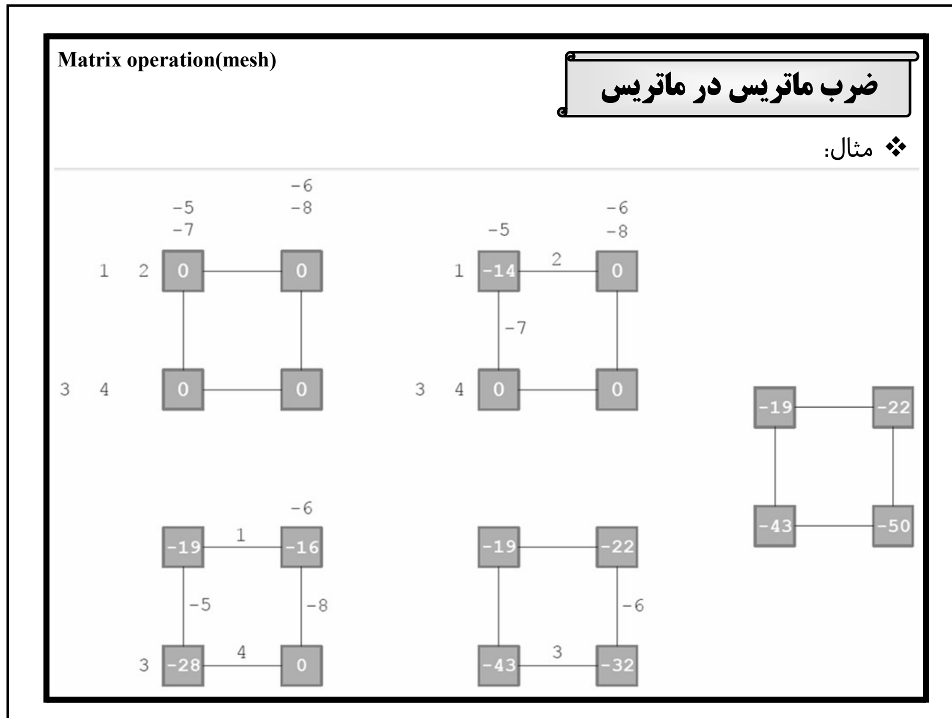
```
procedure MESH MATRIX MULTIPLICATION(A, B, C)
for i = 1 to m do in parallel
        for j = 1 to k do in parallel
                (1) c_ij ← 0
                (2) while P(i, j) receives two inputs a and b do
                        (i) c_ij ← c_ij + (a ×b)
                        (ii) if i < m then
                                send b to P(i+1, j)
                           end if
                        (iii) if j < k then
                                send a to P(i, j+1)
                           end if
                   end while
        end for
end for
```

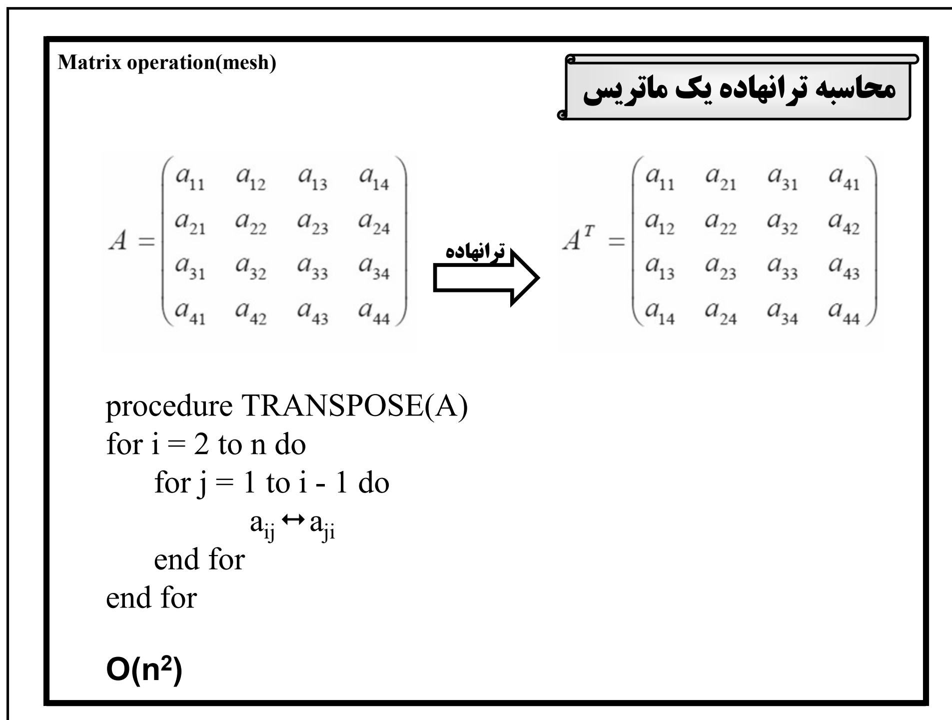
Matrix operation(mesh)

ضرب ماتریس در ماتریس

- ❖ تحلیل:
- ❖ $m \times k$ پردازنده برای ضرب ماتریس $A\ (m \times n)$ در ماتریس $B\ (n \times k)$
- ❖ عناصر a_{m1} و b_{1k} پس از $m + k + n-2$ مرحله به پردازنده $P(m, k)$ می رسند.
- ❖ از آنجا که پردازنده $P(m, k)$ آخرین پردازنده ای است که کار خود را به پایان می رساند، در نتیجه زمان اجرای الگوریتم نیز برابر است با $O(m + k + n)$.
- ❖ با فرض اینکه m و k هر دو کوچکتر یا مساوی n هستند، خواهیم داشت:

- ➢ $t(n) = O(n)$
- ➢ $p(n) = O(n^2)$
- ➢ $c(n) = O(n^3)$





$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{pmatrix}$$

```
procedure TRANSPOSE(A)
for i = 2 to n do
    for j = 1 to i - 1 do
            aij ↔ aji
    end for
end for
```

O(n²)

Matrix operation(mesh)

محاسبه ترانهاده یک ماتریس

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

Matrix Transpose

- پردازنده P(i,j) سه رجیستر دارد:
- A(i, j): برای ذخیره a_{ij} در ابتدا و a_{ji} در انتها
- B(i, j): برای ذخیره مقدار دریافتی از P(i,j+1) یا P(i-1,j)
- C(i, j): برای ذخیره مقدار دریافتی از P(i,j-1) یا P(i+1,j)

Matrix operation(mesh)

محاسبه ترانهاده یک ماتریس

```
procedure MESH TRANSPOSE(A)
Step 1: do steps (1.1) and (1.2) in parallel
        (1.1) for i = 2 to n do in parallel
                  for j = 1 to i – 1 do in parallel
                          C(i-1, j) ← (a_ij, j, i)
                  end for
              end for
        (1.2) for i = 1 to n - 1 do in parallel
                  for j = i + 1 to n do in parallel
                          B(i, j-1) ← (a_ij, j, i)
                  end for
              end for
```

Matrix operation(mesh)

محاسبه ترانهاده یک ماتریس

مثلث پایینی

```
Step 2: do (2.1) and (2.2) and (2.3) in parallel
    (2.1) for i = 2 to n do in parallel
        for j = 1 to i – 1 do in parallel
          while P(i,j) receives input from its neighbours do
            (i) if (a_km, m, k) is received from P(i+1, j) then
                    send it to P(i-1, j)
                end if
            (ii) if (a_km, m, k) is received from P(i-1, j) then
                if (i = m and j = k)then
                        A(i, j) ← a_km {reached destination}
                    else
                        send (a_km, m, k) to P(i+1, j)
                    end if
                end if
          end while
        end for
      end for
```

Matrix operation(mesh)

محاسبه ترانهاده یک ماتریس

قطر اصلی

```
    (2.2) for i = 1 n to do in parallel
        while P(i, i) receives input from its neighbours do
          (i) if (a_km, m, k) is received from P(i+1, i) then
                send it to P(i, i+1)
            end if
        (ii) if (a_km, m, k) is received from P(i, i+1) then
                send it to P(i+1, i)
            end if
        end while
      end for
```

Matrix operation(mesh)

محاسبه ترانهاده یک ماتریس

مثلث بالایی

```
(2.3) for i = 1 n - 1 do in parallel
        for j = i + 1 to n do in parallel
            while P(i, j) receives input from its neighbours do
                (i) if (a_km, m, k) is received from P(i, j+1) then
                        send it to P(i, j-1)
                    end if
                (ii) if (a_km, m, k) is received from P(i, j-1) then
                        if (i = m and j = k) then
                            A(i, j) ← a_km {reached destination}
                        else
                            send (a_km, m, k) to P(i, j+1)
                        end if
                    end if
            end while
        end for
    end for
```

Matrix operation(mesh)

محاسبه ترانهاده یک ماتریس

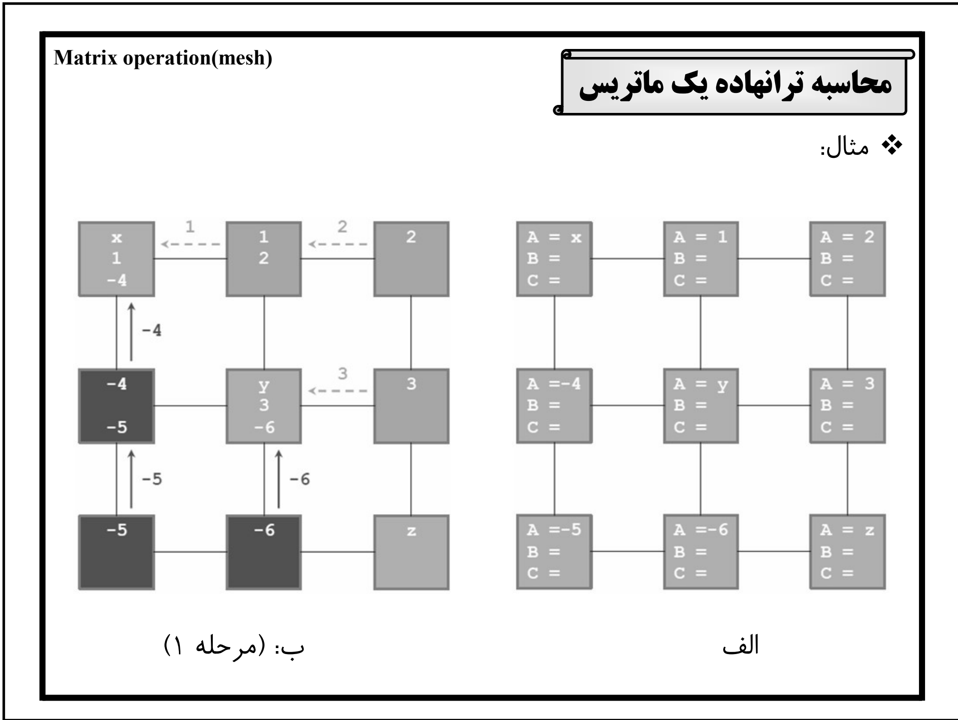
- تحلیل:
- طولانی ترین مسیری که یک عنصر باید طی کند قطر mesh است که حداکثر 2n-2 است.

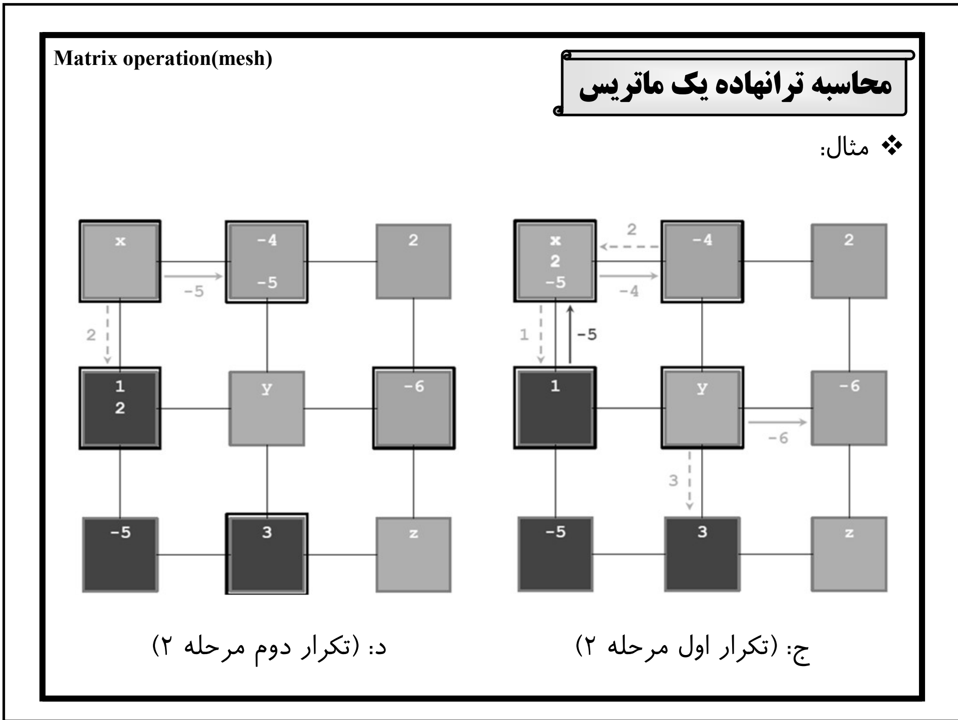
➢ $t(n) = O(n)$

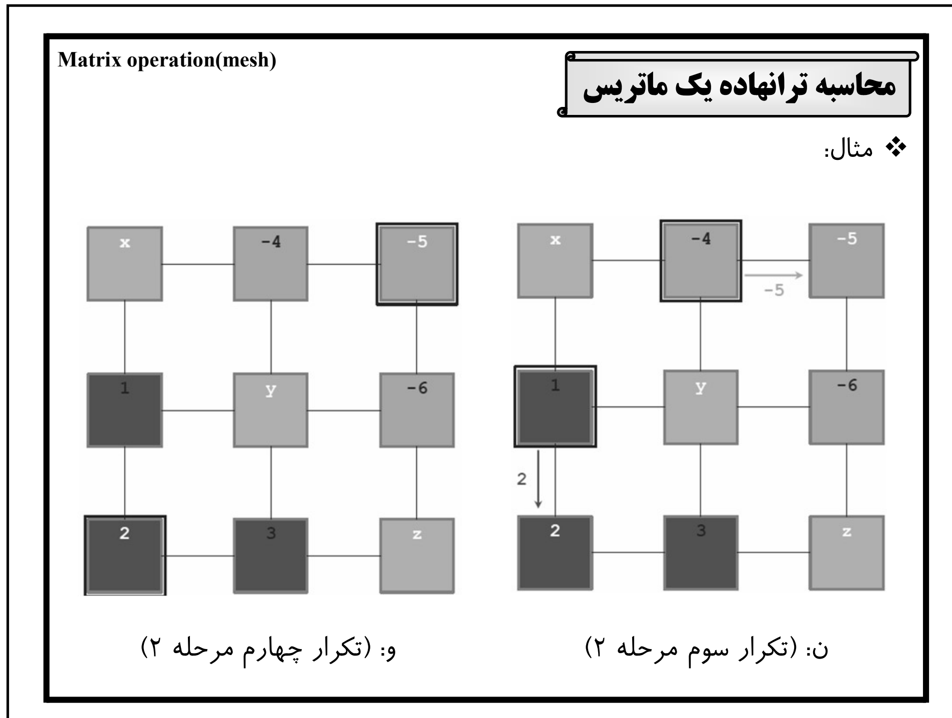
- بنابراین، خواهیم داشت:

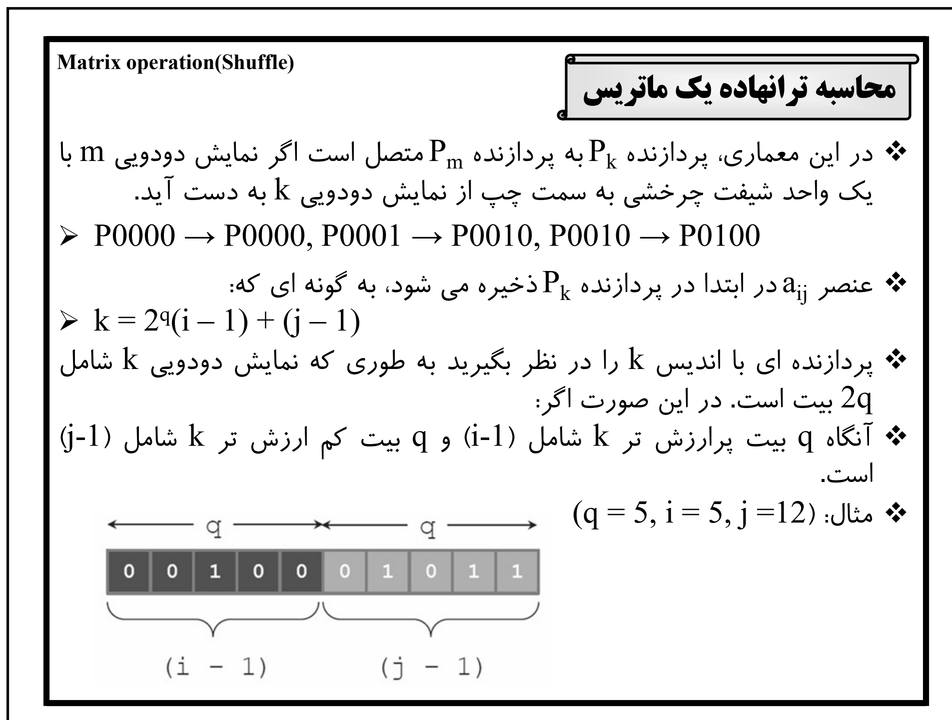
➢ $p(n) = O(n^2)$

➢ $c(n) = O(n^3)$


Matrix operation(mesh)
محاسبه ترانهاده یک ماتریس
❖ مثال:
x
1
-4
1
2
2
-4
-4
-5
y
3
-6
3
3
-5
-6
-5
-6
z
A = x
B =
C =
A = 1
B =
C =
A = 2
B =
C =
A =-4
B =
C =
A = y
B =
C =
A = 3
B =
C =
A =-5
B =
C =
A =-6
B =
C =
A = z
B =
C =
ب: (مرحله ۱)
الف


Matrix operation(mesh)
محاسبه ترانهاده یک ماتریس
❖ مثال:
x
-4
-5
-5
2
2
1
2
y
-6
-5
3
z
x
2
-5
2
-4
-4
2
1
-5
1
y
-6
-6
3
-5
3
z
د: (تکرار دوم مرحله ۲)
ج: (تکرار اول مرحله ۲)

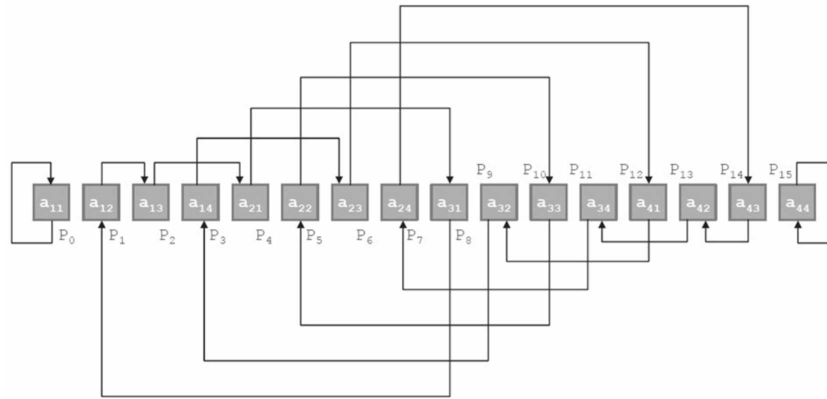

Matrix operation(mesh)
محاسبه ترانهاده یک ماتریس
❖ مثال:
x
-4
-5
1
y
-6
2
3
z
x
-4
-5
-5
1
y
-6
2
2
3
z
و: (تکرار چهارم مرحله ۲)
ن: (تکرار سوم مرحله ۲)


Matrix operation(Shuffle)
محاسبه ترانهاده یک ماتریس
❖ در این معماری، پردازنده P_k به پردازنده P_m متصل است اگر نمایش دودویی m با یک واحد شیفت چرخشی به سمت چپ از نمایش دودویی k به دست آید.
➢ P0000 → P0000, P0001 → P0010, P0010 → P0100
❖ عنصر a_{ij} در ابتدا در پردازنده P_k ذخیره می شود، به گونه ای که:
➢ $k = 2^q(i – 1) + (j – 1)$
❖ پردازنده ای با اندیس k را در نظر بگیرید به طوری که نمایش دودویی k شامل 2q بیت است. در این صورت اگر:
❖ آنگاه q بیت پرارزش تر k شامل (i-1) و q بیت کم ارزش تر k شامل (j-1) است.
❖ مثال: (q = 5, i = 5, j =12)
q
q
0 0 1 0 0 0 1 0 1 1
(i - 1)
(j - 1)

Matrix operation(Shuffle)

محاسبه ترانهاده یک ماتریس

- در این صورت پس از q بار انجام عمل shuffle (شیفت چرخشی به چپ) عنصری که در ابتدا در P_k قرار داشت اکنون در P_s قرار دارد به گونه ای که:
  - $s = 2^q(j-1) + (i-1)$
- به عبارت دیگر، عنصر a_{ij} به مکان عنصر a_{ji} منتقل شده است.

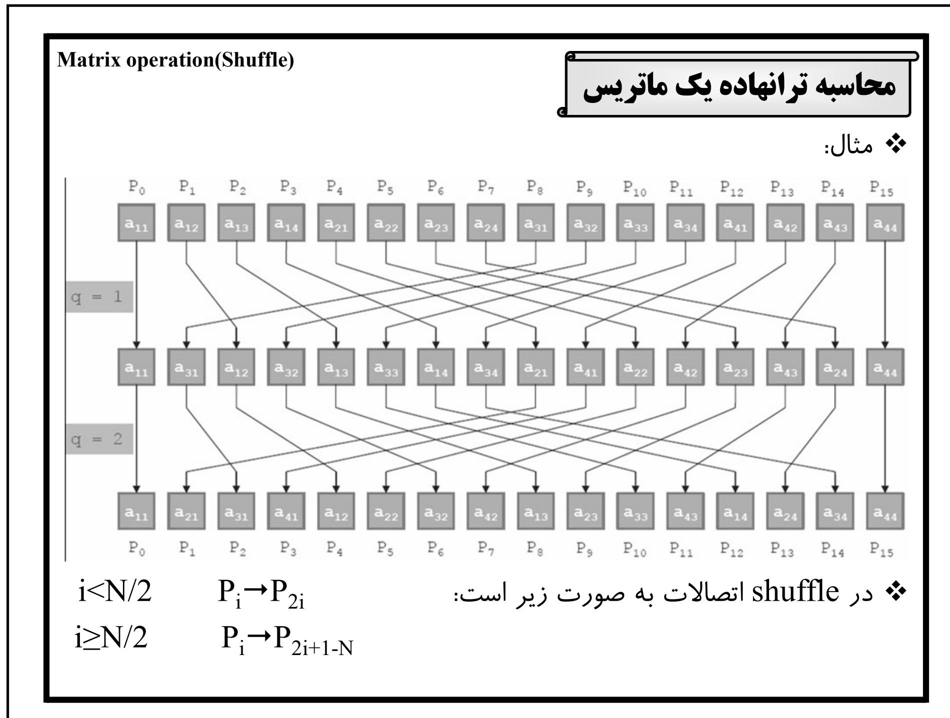


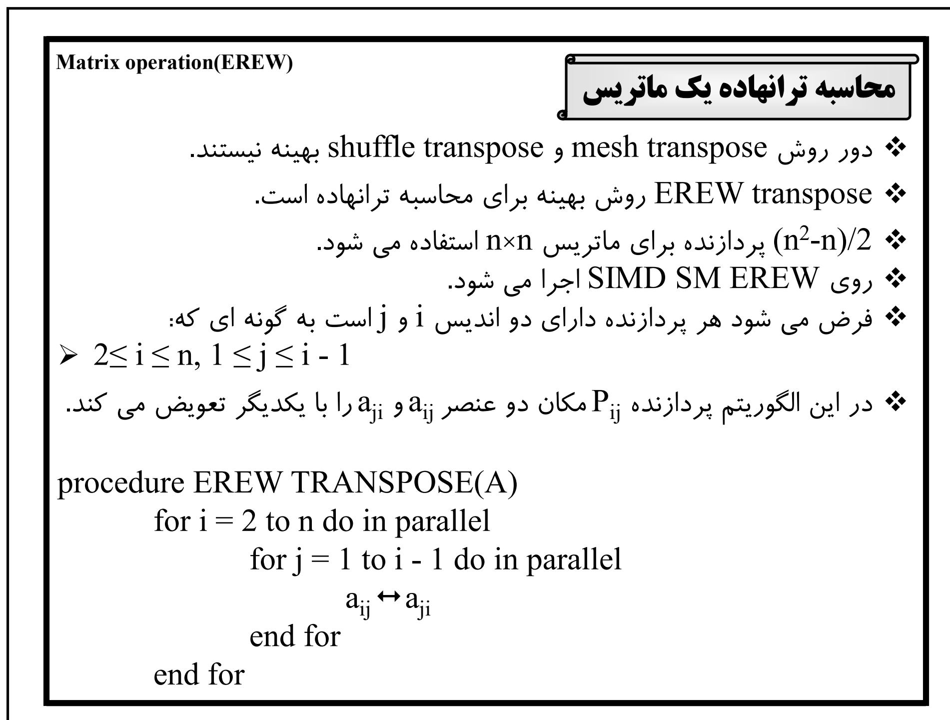
Matrix operation(Shuffle)

محاسبه ترانهاده یک ماتریس

```
procedure SHUFFLE TRANSPOSE(A)
    for i = 1 to q do
        for k = 1 to 2^(2q) – 2 do in parallel
            P_k sends its data to P_(2k mod(2^(2q) – 1))
        end for
    end for
```

- تحلیل:
  - $t(n) = O(\log n)$
  - $p(n) = n^2$
  - $c(n) = O(n^2 \log n)$
- $n=2^q$
- P_k بعد از q عمل shuffle عنصر a_{ji} را خواهد داشت.
- Shuffle از mesh در محاسبه ترانهاده ماتریس سریع تر است.


Matrix operation(Shuffle)
محاسبه ترانهاده یک ماتریس
❖ مثال:
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15
a11 a12 a13 a14 a21 a22 a23 a24 a31 a32 a33 a34 a41 a42 a43 a44
q = 1
a11 a31 a12 a32 a13 a33 a14 a34 a21 a41 a22 a42 a23 a43 a24 a44
q = 2
a11 a21 a31 a41 a12 a22 a32 a42 a13 a23 a33 a43 a14 a24 a34 a44
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15
❖ در shuffle اتصالات به صورت زیر است:
i<N/2 Pi→P2i
i≥N/2 Pi→P2i+1-N


Matrix operation(EREW)
محاسبه ترانهاده یک ماتریس
❖ دور روش mesh transpose و shuffle transpose بهینه نیستند.
❖ EREW transpose روش بهینه برای محاسبه ترانهاده است.
❖ (n2-n)/2 پردازنده برای ماتریس n×n استفاده می شود.
❖ روی SIMD SM EREW اجرا می شود.
❖ فرض می شود هر پردازنده دارای دو اندیس i و j است به گونه ای که:
➢ 2≤ i ≤ n, 1 ≤ j ≤ i - 1
❖ در این الگوریتم پردازنده Pij مکان دو عنصر aij و aji را با یکدیگر تعویض می کند.
procedure EREW TRANSPOSE(A)
for i = 2 to n do in parallel
for j = 1 to i - 1 do in parallel
aij ↔ aji
end for
end for

Matrix operation(EREW)

محاسبه ترانهاده یک ماتریس

❖ تحلیل:

- $t(n) = O(1)$
- $p(n) = (n^2 - n)/2$
- $c(n) = O(n^2)$

Sorting

مرتب سازی

❖ در مرتب سازی ورودی و خروجی به صورت زیر تعریف می شوند.

❖ ورودی:

❖ دنباله ای از n عنصر که بر روی آنها یک ترتیب خطی مانند > تعریف شده است.

- $\mathbf{S} = [s_1, s_2, \ldots, s_n]$

❖ خروجی:

❖ همان دنباله S که عناصر آن به صورت غیر نزولی مرتب شده اند.

- $\mathbf{S'} = [s'_1, s'_2, \ldots, s'_n]\ s'_i < s'_{i+1}, (1 \le i \le n - 1)$

❖ الگوریتم های این بخش:

❖ معماری موازی خاص برای مرتب سازی odd-even merging algorithm

❖ الگوریتم موازی مرتب سازی روی مدل SIMD با معماری linear array

❖ الگوریتم موازی مرتب سازی روی مدل SM SIMD

Sorting(odd-even merging network)

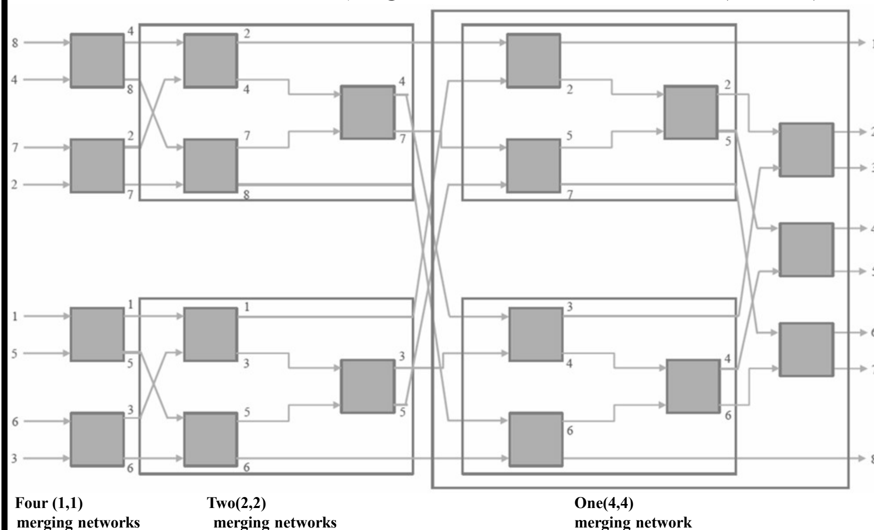
مرتب سازی

- ❖ برای کاربردهای خاص در صنعت استفاده می شود.
- ❖ همه pe ها مقایسه گر هستند. یک پردازشگر و یک بافر دارد.
- ❖ شکل شبکه تقریبا نامنظم (irregular) است.
- ❖ پردازش بسیار سریع است.
- ❖ تعداد pe ها بسیار زیاد است.
- ❖ به دلیل وجود pe ها هزینه سیستم بالا می رود.
- ❖ تعداد ورودیها باید توانی از ۲ باشد.
- ❖ اجرایی شدن الگوریتم برای ورودیهای بیشتر زیر سوال می رود.
- ❖ زمان اجرای الگوریتم از مرتب سازی ترتیبی quicksort سریع تر است.

Sorting(odd-even sorting network)

مرتب سازی

- ❖ برای کاربردهای خاص در صنعت استفاده می شود.



Sorting(odd-even sorting network)

مرتب سازی

❖ تحلیل:

- $t(n)=O(\log^2 n)$
- $p(n)=O(n\log^2 n)$
- $c(n)=O(n\log^4 n)$

❖ log n، مرحله (stage) داریم. در هر stage تعداد دنباله های ادغام شده دو برابر مرحله قبل است.

❖ زمان و تعداد مقایسه کننده های مورد نیاز در مرحله i ام برای ادغام دو دنباله مرتب شده هر کدام 2^{i-1} عنصر را $s(2^i)$ و $q(2^i)$ در نظر بگیرید.

$$\begin{cases} s(2) = 1 & \text{for } i = 1 \\ s(2^i)=s(2^{i-1}) + 1 & \text{for } i > 1 \end{cases}$$

- $s(2^i) = i$
- $t(n) = \sum_{i=1}^{logn} s(2^i) = O(\log^2 n)$

$$\begin{cases} q(2) = 1 & \text{for } i = 1 \\ q(2^i)= 2q(2^{i-1})+ 2^{i-1} - 1 & \text{for } i > 1 \end{cases}$$

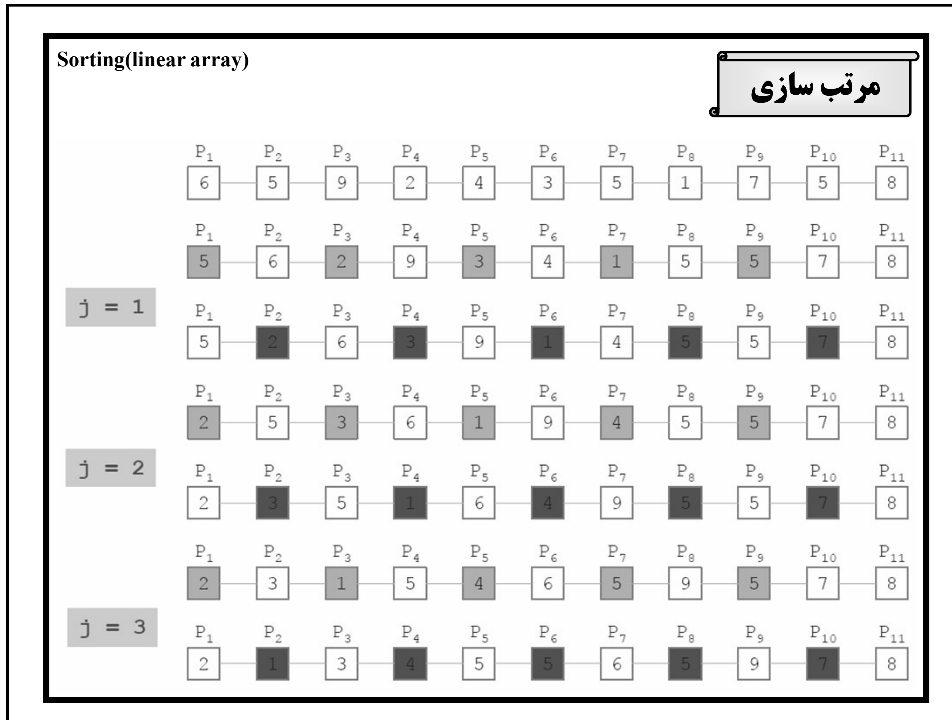
$q(2^i) = (i - 1)2^{i-1} + 1.$

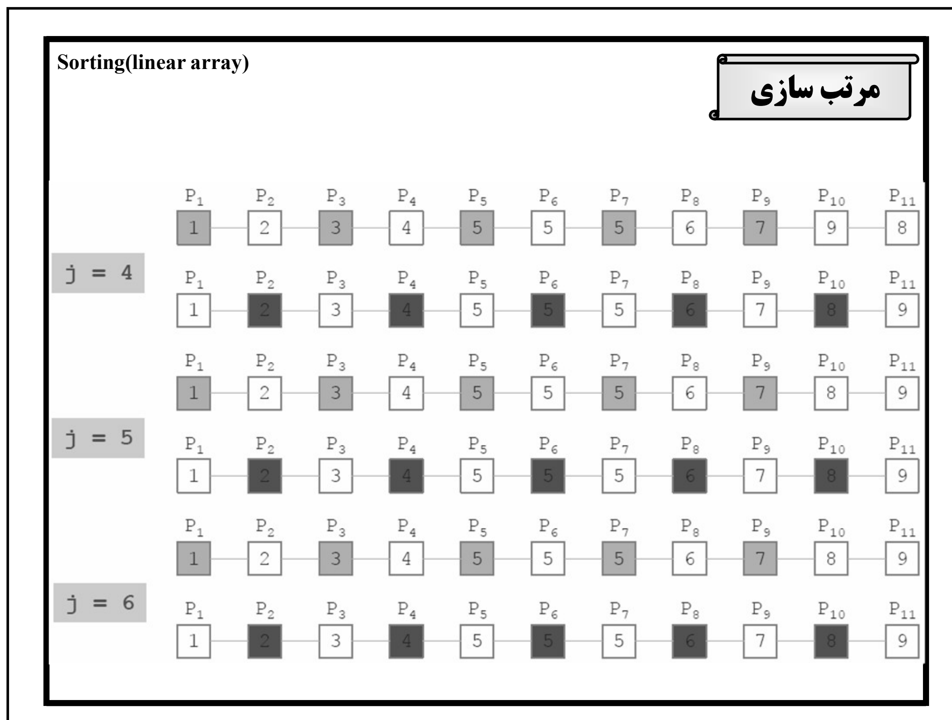
$p(n) = \sum_{i=1}^{logn} q(2^i) = O(n\log^2 n)$

Sorting(linear array)

مرتب سازی

```
procedure ODD-EVEN TRANPOSITION(S)
for j = 1 to n/2 do
(1) for i = 1, 3, …, 2[n/2] - 1 do in parallel
        if x_i > x_{i+1} then
            x_i ↔ x_{i+1}
        end if
    end for
(2) for i = 2, 4, …, 2[(n-1)/2] do in parallel
        if x_i > x_{i+1} then
            x_i ↔ x_{i+1}
        end if
    end for
end for
```


Sorting(linear array)
مرتب سازی
j = 1
j = 2
j = 3


Sorting(linear array)
مرتب سازی
j = 4
j = 5
j = 6

Sorting(linear array)

مرتب سازی

❖ تحلیل:

- $t(n) = O(n)$
- $p(n) = O(n)$
- $c(n) = O(n^2)$

❖ محدودیت: تعداد پردازنده ها منوط به تعداد ورودیها است. n=N

❖ در مرحله اول تمام پردازنده های فرد مقادیر خود را با مقادیر پردازنده بعدی مقایسه کرده و در صورت لزوم عمل جابجایی را انجام می دهند.

❖ در مرحله دوم تمام پردازنده های زوج مقادیر خود را با مقادیر پردازنده بعدی مقایسه کرده و در صورت لزوم عمل جابجایی را انجام می دهند.

❖ این عمل آنقدر تکرار می شود تا داده ها مرتب شوند.

Sorting(linear array)

مرتب سازی

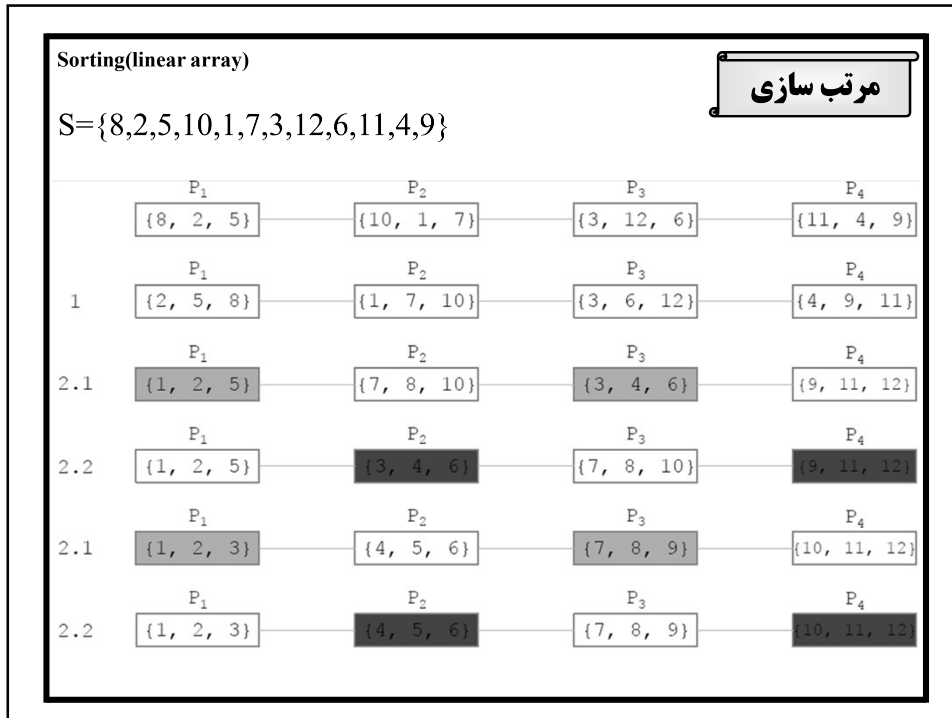
❖ در روش قبل این امکان وجود ندارد که همواره تعداد داده ها و تعداد پردازنده ها برابر باشد بنابراین داده ها به تعداد پردازنده ها تقسیم شوند.

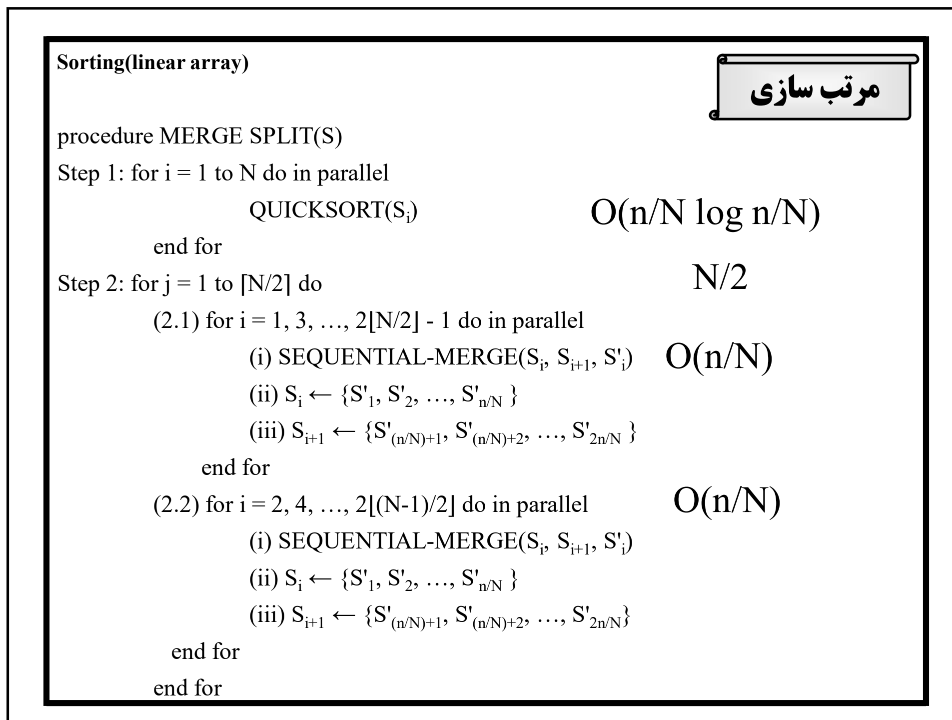
❖ از روش Merge-Split استفاده می شود.

❖ تمام پردازنده های فرد مقادیر خود را با مقادیر پردازنده بعدی Merge کرده، مرتب نموده و عمل Split را انجام می دهند.

❖ سپس تمام پردازنده های زوج مقادیر خود را با مقادیر پردازنده بعدی Merge کرده، مرتب نموده و عمل Split را انجام می دهند.

❖ این عمل آنقدر تکرار می شود تا داده ها مرتب شوند.


Sorting(linear array)
مرتب سازی
S={8,2,5,10,1,7,3,12,6,11,4,9}
P1 P2 P3 P4
{8, 2, 5} {10, 1, 7} {3, 12, 6} {11, 4, 9}
1 {2, 5, 8} {1, 7, 10} {3, 6, 12} {4, 9, 11}
2.1 {1, 2, 5} {7, 8, 10} {3, 4, 6} {9, 11, 12}
2.2 {1, 2, 5} {3, 4, 6} {7, 8, 10} {9, 11, 12}
2.1 {1, 2, 3} {4, 5, 6} {7, 8, 9} {10, 11, 12}
2.2 {1, 2, 3} {4, 5, 6} {7, 8, 9} {10, 11, 12}


Sorting(linear array)
مرتب سازی
procedure MERGE SPLIT(S)
Step 1: for i = 1 to N do in parallel
QUICKSORT(S_i)
O(n/N log n/N)
end for
Step 2: for j = 1 to ⌈N/2⌉ do
N/2
(2.1) for i = 1, 3, …, 2⌊N/2⌋ - 1 do in parallel
(i) SEQUENTIAL-MERGE(S_i, S_{i+1}, S'_i)
O(n/N)
(ii) $S_i \leftarrow \{S'_1, S'_2, \ldots, S'_{n/N}\}$
(iii) $S_{i+1} \leftarrow \{S'_{(n/N)+1}, S'_{(n/N)+2}, \ldots, S'_{2n/N}\}$
end for
(2.2) for i = 2, 4, …, 2⌊(N-1)/2⌋ do in parallel
O(n/N)
(i) SEQUENTIAL-MERGE(S_i, S_{i+1}, S'_i)
(ii) $S_i \leftarrow \{S'_1, S'_2, \ldots, S'_{n/N}\}$
(iii) $S_{i+1} \leftarrow \{S'_{(n/N)+1}, S'_{(n/N)+2}, \ldots, S'_{2n/N}\}$
end for
end for

Sorting(linear array)

مرتب سازی

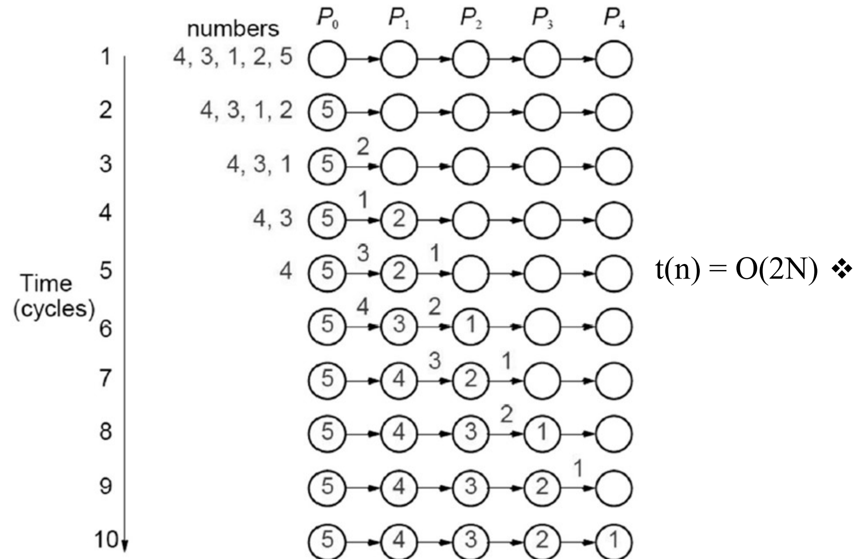
❖ تحلیل:

- $t(n) = O(n/N \log n/N) + N/2 \times O(n/N) = O((n \log n)/N) + O(n)$
- $p(n) = O(N)$
- $c(n) = O(n \log n) + O(n \times N)$

❖ در نتیجه این الگوریتم به ازای $N \leq \log n$ هزینه بهینه است.

Sorting(Insertion Sort)

مرتب سازی



❖ $t(n) = O(2N)$

Sorting(Shear Sort on Mesh)

مرتب سازی

```
procedure SHEAR SORT
for i=1 to 2log n+1 do
  if (i is odd) then
        if odd(row) then  SortLeftToRight
        if even(row) SortRightToLeft
 else
        SortTopToBottom
```

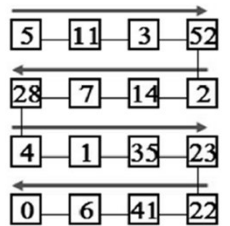
O(log n)

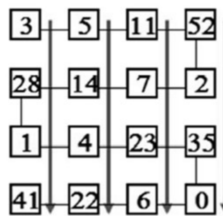
- $t(n)=O(\log n)$
- $p(n)=O(n^2)$
- $c(n)=O(n^2 \log n)$

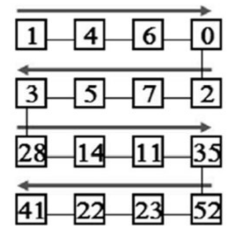
Sorting(Shear Sort on Mesh)

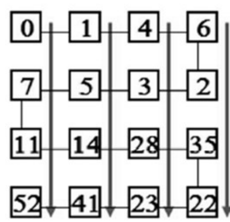
مرتب سازی

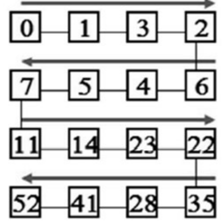
Snake like Sorting

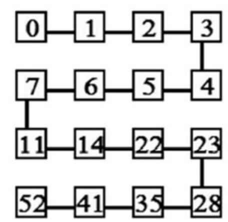












Sorting(SM SIMD)

مرتب سازی

❖ مرتب سازی بر روی مدل CRCW است. تعداد داده ها n و pe ها n^2 است.

❖ بحث write strategy در مدل CRCW مطرح است.

```
procedure CRCW SORT(S)
for i = 1 to n do in parallel
        for j = 1 to n do in parallel
                if (s_i > s_j) or (s_i = s_j and i > j) then
                        P(i, j) adds 1 in c_i
                else
                        P(i, j) adds 0 in c_i
                end if
        end for
end for
for i = 1 to n do in parallel
        P(i, 1) stores s_i in position 1 + c_i of  S
end for
```

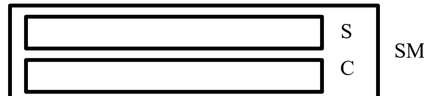
Sorting(SM SIMD)

مرتب سازی

S					C	S
	P(1,1)	P(1,2)	P(1,3)	P(1,4)		
5	5, 5	5, 2	5, 4	5, 5	2	2
	P(2,1)	P(2,2)	P(2,3)	P(2,4)		
2	2, 5	2, 2	2, 4	2, 5	0	4
	P(3,1)	P(3,2)	P(3,3)	P(3,4)		
4	4, 5	4, 2	4, 4	4, 5	1	5
	P(4,1)	P(4,2)	P(4,3)	P(4,4)		
5	5, 5	5, 2	5, 4	5, 5	3	5

تحلیل :

- $t(n) = O(1)$
- $p(n) = O(n^2)$
- $c(n) = O(n^2)$



SM

Searching

جستجو

- ❖ ورودی:
- ❖ دنباله ای از n عدد صحیح و عدد صحیح x.

➢ $\mathbf{S} = [s_1, s_2, \ldots, s_n]$

- ❖ خروجی:
- ❖ اندیس عنصر x در لیست S (در صورت وجود)

➢ $\mathbf{s_k = x}$

- ❖ جستجوی موازی بر روی یک آرایه مرتب
- ❖ مدل EREW
- ❖ مدل CRCW
- ❖ مدل CREW
- ❖ جستجو بر روی یک آرایه نامرتب
- ❖ جستجو روی مدل SIMD با ساختار tree
- ❖ جستجو روی مدل SIMD با ساختار mesh

Searching(EREW on sorted list)

جستجو

- ❖ فرض: همه عناصر S متمایز هستند.
- ❖ الگوریتم جستجو بر روی EREW:
  - ❖ انتشار مقدار x در بین پردازنده ها: O(log N)
  - ❖ N تعداد پردازنده ها و n تعداد داده ها
  - ❖ تقسیم S به N قسمت مساوی و توزیع قسمتها در بین پردازنده ها
  - ❖ اجرای همزمان جستجوی دودویی (binary search) به وسیله پردازنده ها بر روی قسمتهای مربوطه: O(log(n/N))
- ❖ زمان اجرا:

➢ $O(\log N) + O(\log(n/N)) = O(\log n)$

Searching(CREW on sorted list)

جستجو

- ❖ فرض: همه عناصر S متمایز هستند.
- ❖ الگوریتم جستجو بر روی CREW:
  - ❖ N تعداد پردازنده ها و n تعداد داده ها
  - ❖ تقسیم S به N قسمت مساوی و توزیع قسمتها در بین پردازنده ها
  - ❖ اجرای همزمان جستجوی دودویی (binary search) به وسیله پردازنده ها بر روی قسمتهای مربوطه: O(log(n/N))
- ❖ زمان اجرا:

➢ O(log(n/N))

Searching(CRCW on sorted list)

جستجو

- ❖ اگر عناصر S متمایز نباشند.
- ❖ ممکن است چند پردازنده به طور همزمان x را یافته و بخواهند مقدار k را تغییر دهند.
- ❖ باید از CRCW استفاده شود.
- ❖ بستگی به استراتژی write دارد. O(log N)
- ❖ زمان اجرا:

➢ O(log(n/N)) + O(log N) = O(log n)

Searching(SM)

جستجو

```
procedure SM (S, x, k)
Step 1: for i = 1 to N do in parallel
                read x
        end for
Step 2: for i = 1 to N do in parallel
        (2.1) S_i ← {s_(i-1)(n/N)+1, s_(i-1)(n/N)+2, …, s_i(n/N)}
        (2.2) SEQUENTIAL SEARCH(S_i, x, k_i)
        end for
Step 3: for i = 1 to N do in parallel
                if k_i > 0 then
                        k ← k_i
                end if
        end for
```

Searching(SM-EREW)

جستجو

❖ مرحله ۱) با استفاده از الگوریتم انتشار

➢ O(log N)

❖ مرحله ۲) جستجوی ترتیبی

➢ O(n/N)

❖ مرحله ۳) با استفاده از رویه STORE و یک مکانیزم مناسب برای رفع تصادم میان نوشتن های همزمان

➢ O(log N)

❖ تحلیل:

➢ $t(n) = O(\log N) + O(n/N)$

➢ $c(n)= O(N \log N) + O(n)$

Searching(SM-ERCW)

جستجو

❖ مرحله ۱) با استفاده از الگوریتم انتشار

➢ O(log N)

❖ مرحله ۲) جستجوی ترتیبی

➢ O(n/N)

❖ مرحله ۳) در زمان ثابت

❖ تحلیل:

➢ t(n) = O(log N) + O(n/N)

➢ c(n)= O(N log N) + O(n)

Searching(SM-CREW)

جستجو

❖ مرحله ۱) در زمان ثابت

❖ مرحله ۲) جستجوی ترتیبی

➢ O(n/N)

❖ مرحله ۳) با استفاده از رویه STORE و یک مکانیزم مناسب برای رفع تصادم میان نوشتن های همزمان

➢ O(log N)

❖ تحلیل:

➢ t(n) = O(log N) + O(n/N)

➢ c(n)= O(N log N) + O(n)

Searching(SM-CRCW)

جستجو

❖ مرحله ۱) در زمان ثابت

❖ مرحله ۲) جستجوی ترتیبی

➢ $O(n/N)$

❖ مرحله ۳) در زمان ثابت

❖ تحلیل:

➢ $t(n) = O(n/N)$

➢ $c(n)= O(n)$

Searching(on tree)

جستجو

❖ استفاده از مدل SIMD با اتصال درختی شامل n گره برگ برای جستجو در یک فایل حاوی n رکورد

❖ الگوریتم Querying:

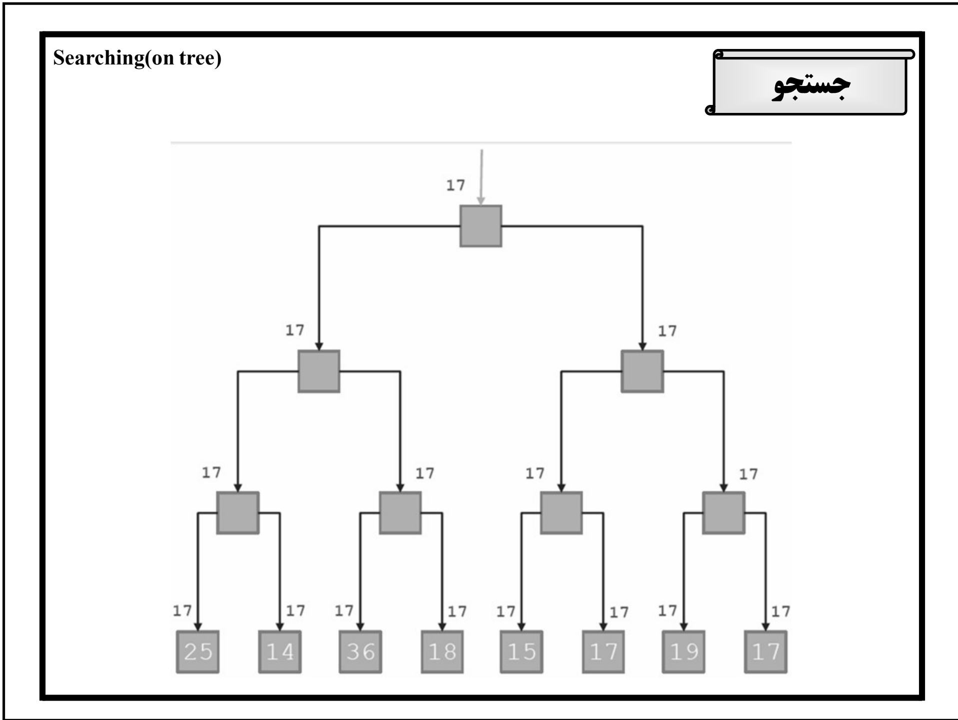
❖ مرحله ۱) خواندن x به وسیله گره ریشه و انتشار آن به سمت پایین تا رسیدن به برگ های درخت

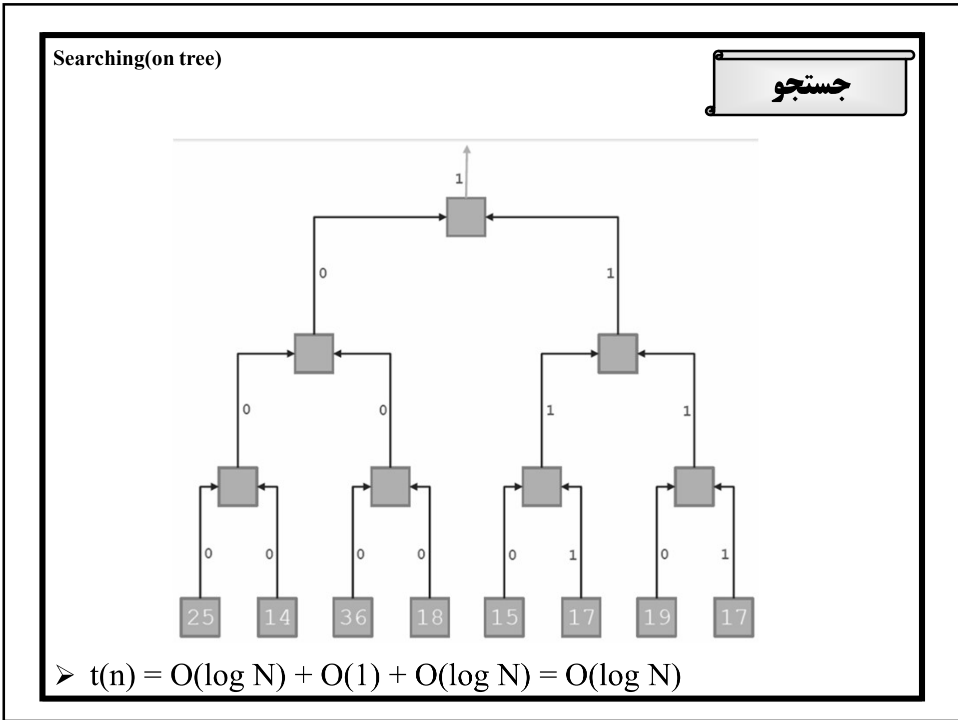
❖ مرحله ۲) مقایسه همزمان مقدار هر برگ با مقدار x

❖ تولید ۱ در صورت مساوی بودن مقادیر

❖ تولید ۰ در صورت عدم تساوی

❖ مرحله ۳) محاسبه OR گره های برگ در گره والد و تکرار این عمل در سطوح بالاتر درخت تا رسیدن به گره ریشه


Searching(on tree)
جستجو
17
17
17
17
17
17
17
17
17
17
17
17
17
17
17
25
14
36
18
15
17
19
17


Searching(on tree)
جستجو
1
0
1
0
0
1
1
0
0
0
0
0
1
0
1
25
14
36
18
15
17
19
17
t(n) = O(log N) + O(1) + O(log N) = O(log N)

Searching(on tree)

جستجو

❖ اجرای q Query با استفاده از pipelining:

❖ پردازش پرس و جوی اول:

➢ O(log n)

❖ پردازش پرس و جوی دوم:

➢ O(log n) + 1

❖ پردازش پرس و جوی q:

➢ O(log n) + (q – 1)

Searching(on tree)

جستجو

❖ الگوریتم Counting:

❖ برگرداندن تعداد عناصر برابر با x. همانند الگوریتم قبل است با این تفاوت که به جای محاسبه OR فرزندان خود، مجموع مقدار آن ها را برمی گردانند.

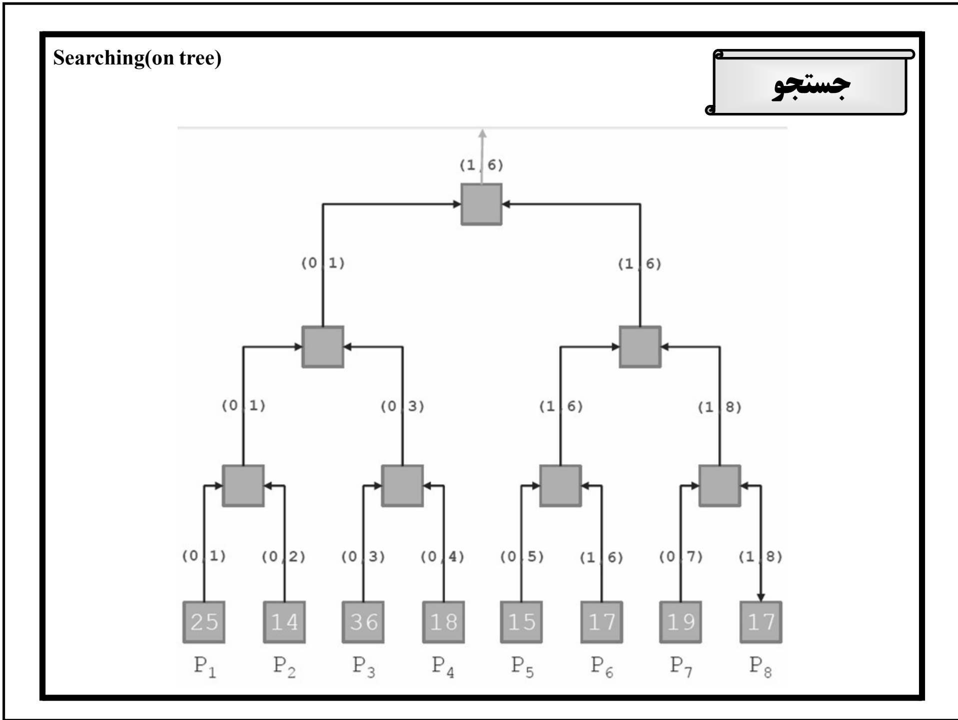
❖ الگوریتم Positioning:

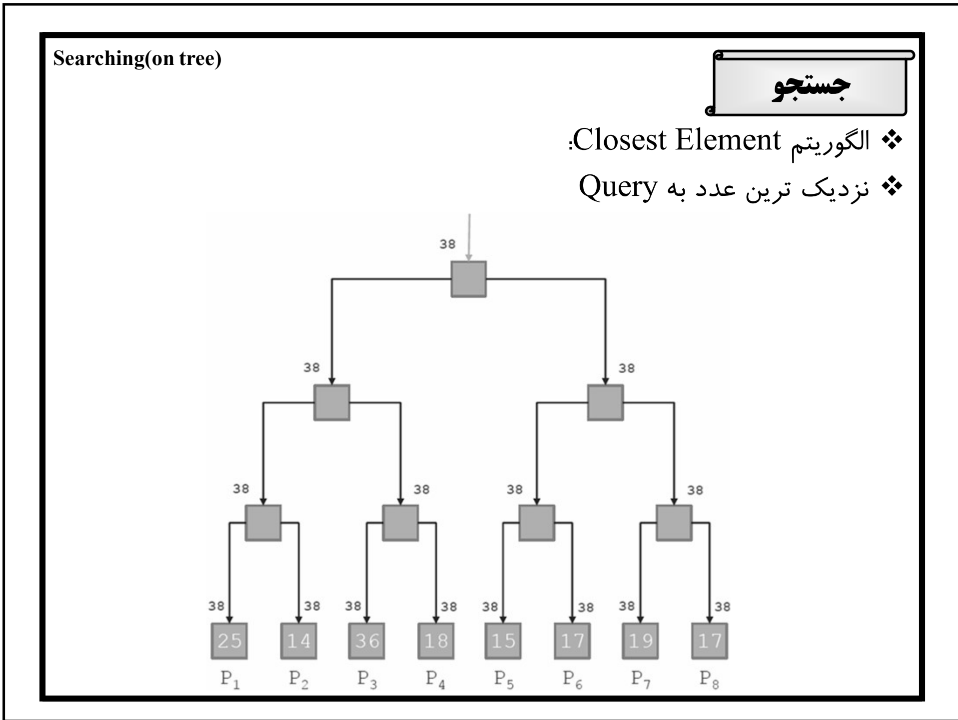
❖ مرحله ۱) خواندن x به وسیله گره ریشه و انتشار آن به سمت پایین تا رسیدن به برگ های درخت

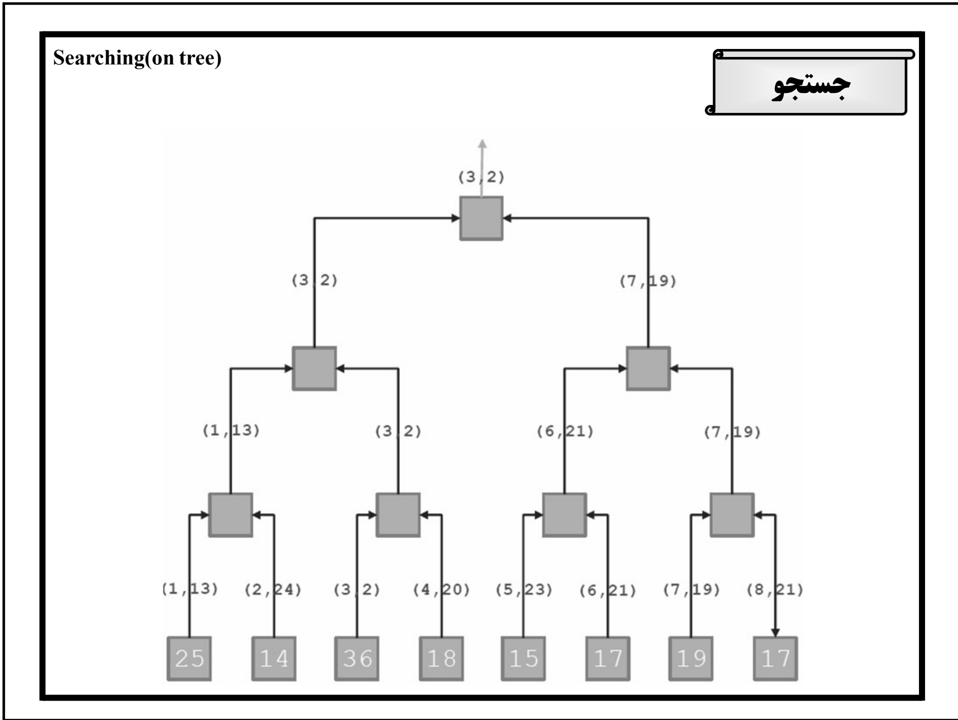
❖ مرحله ۲) مقایسه همزمان مقدار هر برگ با مقدار x

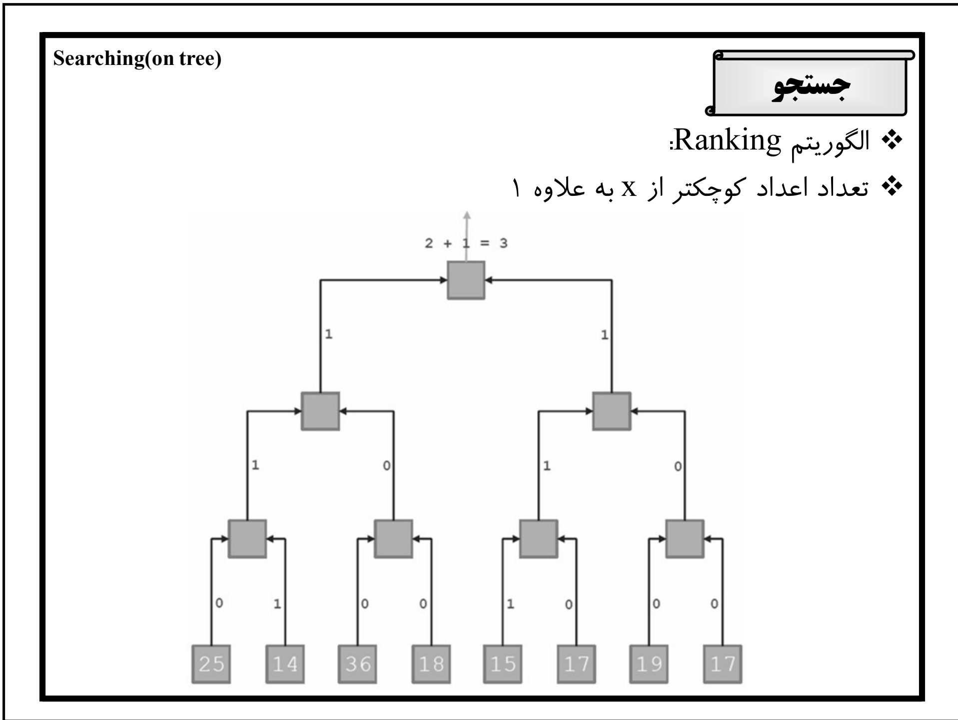
- ❖ تولید (index،۱) در صورت مساوی بودن مقادیر
- ❖ تولید (index،۰) در صورت مساوی نبودن مقادیر

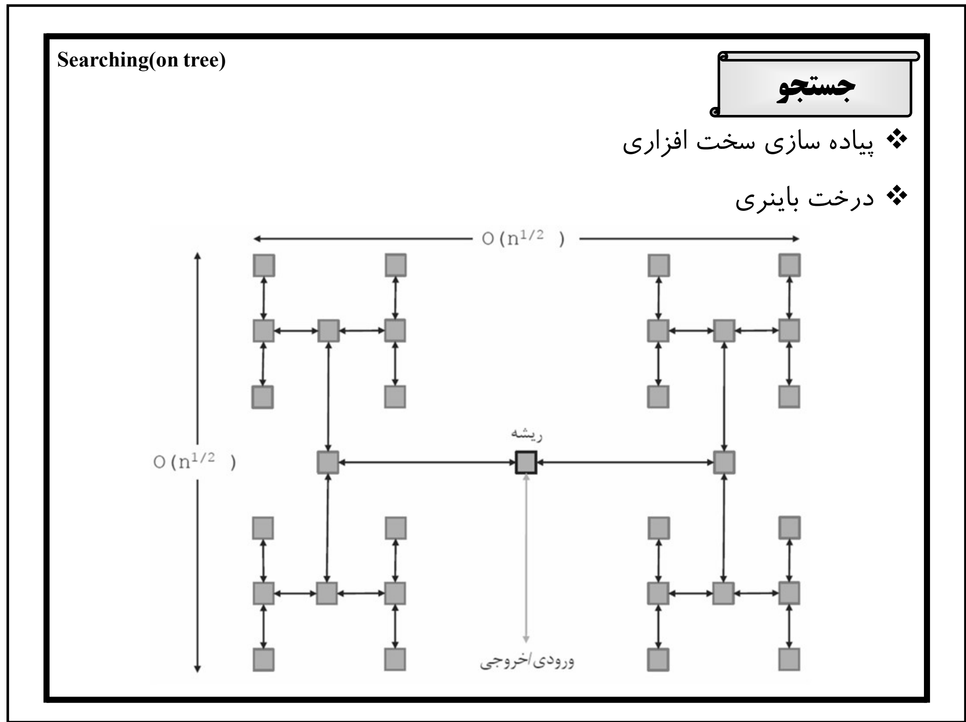
❖ مرحله ۳) ارسال زوجی که یک را به عنوان عنصر اول داد به بالا. اولویت: ارسال از سمت چپ مقدم است.

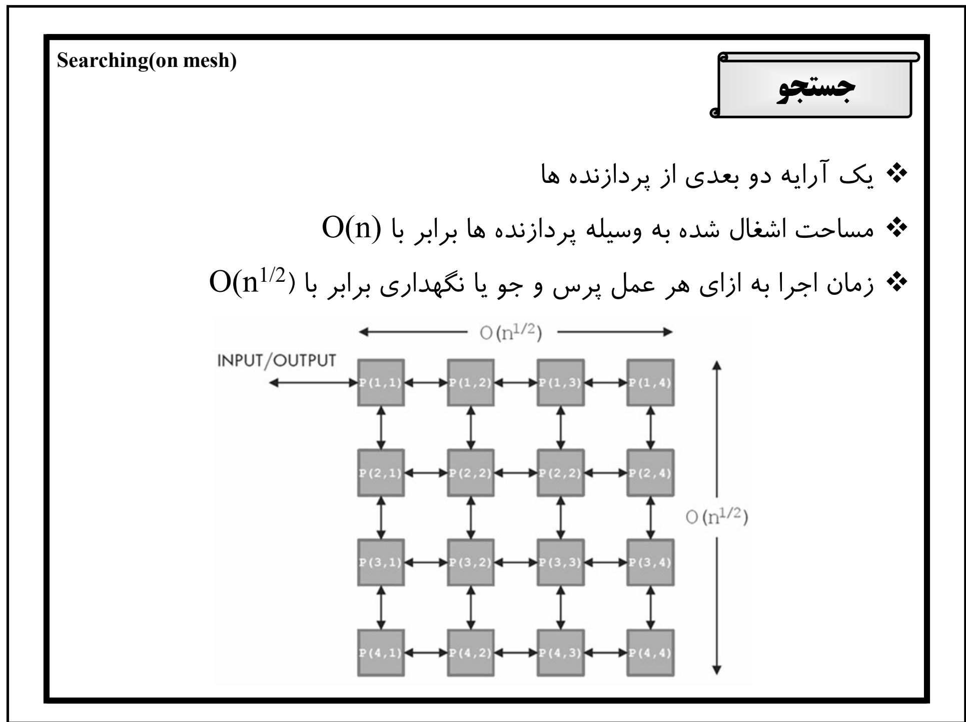

Searching(on tree)
جستجو
(1,6)
(0,1)
(1,6)
(0,1)
(0,3)
(1,6)
(1,8)
(0,1)
(0,2)
(0,3)
(0,4)
(0,5)
(1,6)
(0,7)
(1,8)
25
14
36
18
15
17
19
17
P1
P2
P3
P4
P5
P6
P7
P8


Searching(on tree)
جستجو
❖ الگوریتم Closest Element:
❖ نزدیک ترین عدد به Query
38
38
38
38
38
38
38
38
38
38
38
38
38
38
38
25
14
36
18
15
17
19
17
P1
P2
P3
P4
P5
P6
P7
P8


Searching(on tree)
جستجو
(3,2)
(3,2)
(7,19)
(1,13)
(3,2)
(6,21)
(7,19)
(1,13)
(2,24)
(3,2)
(4,20)
(5,23)
(6,21)
(7,19)
(8,21)
25
14
36
18
15
17
19
17


Searching(on tree)
جستجو
❖ الگوریتم Ranking:
❖ تعداد اعداد کوچکتر از x به علاوه ۱
2 + 1 = 3
1
1
1
0
1
0
0
1
0
0
1
0
0
0
25
14
36
18
15
17
19
17


Searching(on tree)
جستجو
❖ پیاده سازی سخت افزاری
❖ درخت باینری
O(n1/2)
O(n1/2)
ریشه
ورودی/خروجی


Searching(on mesh)
جستجو
❖ یک آرایه دو بعدی از پردازنده ها
❖ مساحت اشغال شده به وسیله پردازنده ها برابر با O(n)
❖ زمان اجرا به ازای هر عمل پرس و جو یا نگهداری برابر با O(n1/2)
O(n1/2)
INPUT/OUTPUT
O(n1/2)
P(1,1)
P(1,2)
P(1,3)
P(1,4)
P(2,1)
P(2,2)
P(2,2)
P(2,4)
P(3,1)
P(3,2)
P(3,3)
P(3,4)
P(4,1)
P(4,2)
P(4,3)
P(4,4)

Searching(on mesh)

جستجو

```
procedure MESH-SEARCH(S, x, answer)
Step 1: { P(1,1) READS THE INPUT }
    if x = s_{1,1} then b_{1,1} ← 1
               else b_{1,1} ← 0                              O(1)
Step 2: { UNFOLDING }
    for i = 1 to n^{1/2} – 1 do
    (2.1) for j = 1 to i do in parallel
          (i) P(j, i) transmits (b_{j,i}, x) to P(j, i + 1)
          (ii) if (b_{j,i} = 1 or x = s_{j,i+1}) then b_{j,i+1} ← 1
                                          else b_{j,i+1} ← 0
             end if
          end for                                            O(n^{1/2})
    (2.2) for j = 1 to i + 1 do in parallel
          (i) P(i, j) transmits (b_{i,j}, x) to P(i + 1, j)
          (ii) if (b_{i,j} = 1 or x = s_{i+1,j}) then b_{i+1,j} ← 1
                                          else b_{i+1,j} ← 0
             end if
          end for
    end for
```

Searching(on mesh)

جستجو

```
Step 3: { FOLDING }
    for i = n^{1/2} downto 2 do
    (3.1) for j = 1 to i do in parallel
              P(j, i) transmits b_{j,i} to P(j, i - 1)
          end for
    (3.2) for j = 1 to i -1 do in parallel
              b_{j,i-1} ← b_{j,i}
          end for
    (3.3) if (b_{i,i-1} = 1 or b_{i,i} = 1) then b_{i,i-1} ← 1
                                    else  b_{i,i-1} ← 0
          end if                                             O(n^{1/2})
    (3.4) for j = 1 to i - 1 do in parallel
              P(i, j) transmits b_{i,j} to P(i – 1, j)
          end for
    (3.5) for j = 1 to i - 2 do in parallel
              b_{i-1, j} ← b_{i,j}
          end for
    (3.6) if (b_{i-1,i-1} = 1 or b_{i,i-1} = 1) then b_{i-1,i-1} ← 1
                                        else  b_{i-1,i-1} ← 0
          end if
    end for
Step 4: { P(1,1) PRODUCES THE OUTPUT }
    if b_{1,1} = 1 then answer ← yes
                  else answer ← no
    end if                                                   O(1)
```

Searching(on mesh)

جستجو

❖ مرحله ۱و ۴)

➢ O(1)

❖ مرحله ۲و۳)

➢ $O(n^{1/2})$

❖ Folding & UnFolding

❖ انتشار x از چپ به راست و از بالا به پایین

❖ انتشار بیت خروجی از راست به چپ و از پایین به بالا

❖ زمان اجرا:

➢ $t(n) = O(n^{1/2})$

Searching(on mesh)

جستجو

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

→

15			

15	15		

15	15		
15	15		

15	15	15	
15	15	15	

15	15	15	
15	15	15	
15	15	15	

15	15	15	15
15	15	15	15
15	15	15	15

15	15	15	15
15	15	15	15
15	15	15	15
15	15	15	15

Searching(on mesh)

جستجو

YES

Graph Theory

تئوری گراف

- ❖ گراف شامل مجموعه ای متناهی از گره ها و یالها (لبه ها) است.
- ❖ جزء مسائل پایه ای در علوم است.
- ❖ از ماتریس برای ذخیره سازی گراف در کامپیوتر استفاده می شود.
- ❖ گراف می تواند جهت دار باشد. گراف جهت دار به صورت زیر تعریف می شود.

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is connected to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$

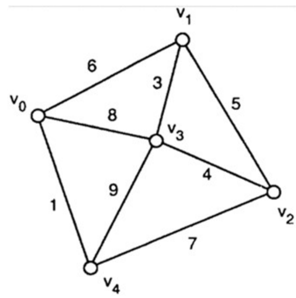
a b c d

$$\begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Graph Theory

تئوری گراف

❖ گراف وزن دار (weighted graph)



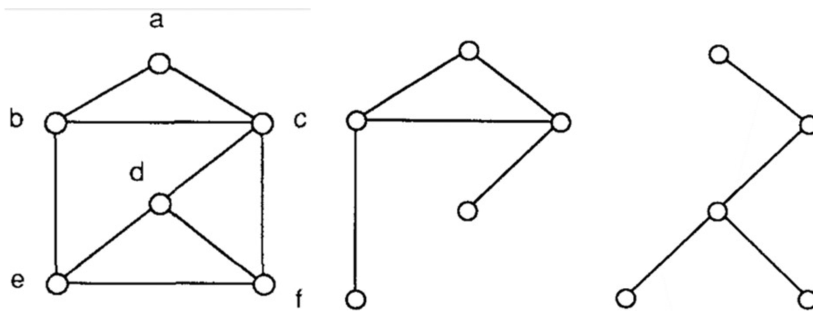
	0	1	2	3	4
0	0	6	0	8	1
1	6	0	5	3	0
2	0	5	0	4	7
3	8	3	4	0	9
4	1	0	7	9	0

❖ گراف وزن دار هم می تواند جهت دار (undirected) باشد.

Graph Theory

تئوری گراف

❖ زیرگراف (subgraph)



❖ زیرگراف G گرافی است که تمام یالها و گره های آن در G وجود دارد.

Graph Theory(Connectivity)

تئوری گراف

- Connectivity Matrix
- برای یک گراف با n گره ماتریس n×n، C با مولفه های زیر تعریف می شود.

$$c_{jk} = \begin{cases} 1 & \text{if there is a path of length 0 or more from } v_j \text{ to } v_k, \\ 0 & \text{otherwise,} \end{cases}$$

- این امکان وجود دارد که از مبدا از طریق چند گره به مقصد مسیر وجود داشته باشد.
- وزن در این حالت مهم نیست.
- به این ماتریس Reflexive و Transitive هم گفته می شود.

Graph Theory(Connectivity)

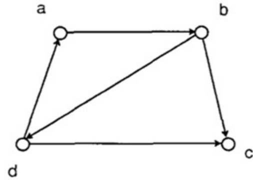
تئوری گراف

```
procedure CUBE CONNECTIVITY (A, C)
Step 1:   for j= to n-1 do in parallel
              A(0,j,j) ← 1
          end for
Step 2:   for j=0 to n-1 do in parallel
               for k =0 to n-1 do in parallel
                    B(0,j,k) ← A(0,j,k)
               end for
          end for
Step 3:   for i = 1 to log n-1 do
              (3.1) CUBE MATRIX MULTIPLICATION (A, B, C)
              (3.2) for j=0  to n-1 do in parallel
                        for k =0 to n-1 do in parallel
                              A(0,j,k) ← C(0,j,k)
                              B(0,j,k) ← C(0,j,k)
                        end for
                   end for
          end for
```

$O(\log^2 n)$

Graph Theory(Connectivity)

تئوری گراف



$$\begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix} \qquad B^2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

❖ تحلیل:

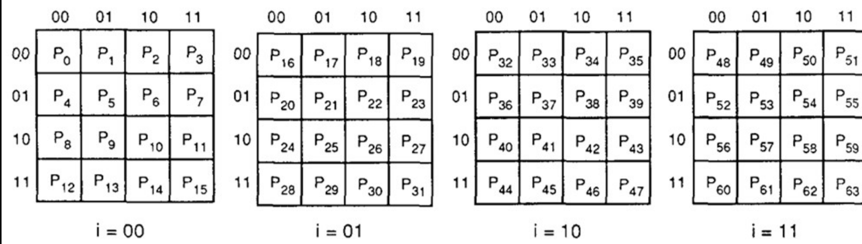
- CUBE MATRIX MULTIPLICATION $O(\log n)$
- $t(n) = O(\log^2 n)$
- $p(n) = O(n^3)$
- $c(n) = O(n^3 \log^2 n)$

Graph Theory(Connectivity)

تئوری گراف

❖ CUBE MATRIX MULTIPLICATION

$$A = \begin{bmatrix} 17 & 23 & 27 & 3 \\ 9 & 1 & 14 & 16 \\ 31 & 26 & 22 & 8 \\ 15 & 4 & 10 & 29 \end{bmatrix} \qquad B = \begin{bmatrix} -7 & -25 & -19 & -5 \\ -18 & -30 & -28 & -12 \\ -13 & -21 & -11 & -32 \\ -20 & -2 & -6 & -24 \end{bmatrix}$$



Graph Theory(Connectivity)

تئوری گراف

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

17 -7	17 -25	17 -19	17 -5
9 -7	9 -25	9 -19	9 -5
31 -7	31 -25	31 -19	31 -5
15 -7	15 -25	15 -19	15 -5

23 -18	23 -30	23 -28	23 -12
1 -18	1 -30	1 -28	1 -12
26 -18	26 -30	26 -28	26 -12
4 -18	4 -30	4 -28	4 -12

27 -13	27 -21	27 -11	27 -32
14 -13	14 -21	14 -11	14 -32
22 -13	22 -21	22 -11	22 -32
10 -13	10 -21	10 -11	10 -32

3 -20	3 -2	3 -6	3 -24
16 -20	16 -2	16 -6	16 -24
8 -20	8 -2	8 -6	8 -24
29 -20	29 -2	29 -6	29 -24

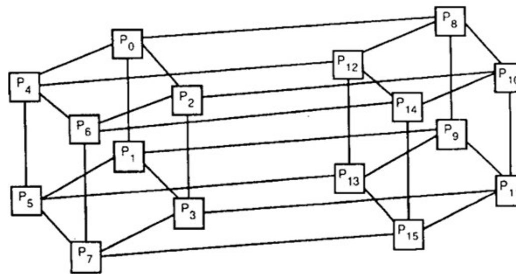
Graph Theory(Connectivity)

تئوری گراف

❖ CUBE MATRIX MULTIPLICATION

Result

-944	-1688	-1282	-1297
-583	-581	-449	-889
-1131	-2033	-1607	-1363
-887	-763	-681	-1139



❖ تحلیل:

- $t(n) = O(\log n)$
- $p(n) = O(n^3)$
- $c(n) = O(n^3 \log n)$

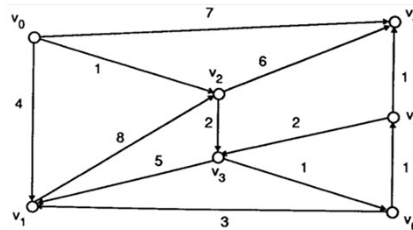
Graph Theory(Shortest Path)

تئوری گراف

- ❖ Shortest Path
- ❖ هدف یافتن کوتاهترین مسیر به ازاء هر جفت v_i و v_j در گراف است.
- ❖ d_{ij}^k: کوتاهترین مسیر از v_i به v_j که از حداکثر k-1 گره واسط عبور کرده باشد.
- ❖ $d_{ij}^k = w_{ij}$، اگر چنین فاصله ای نباشد، $d_{ij}^k = \infty$.
- ❖ از ضرب ماتریس استفاده می شود که عملگرها متفاوت هستند.

❖ $Sp(v_0,v_4)$:

$(v_0,v_2),(v_2,v_3),(v_3,v_6),(v_6,v_5),(v_5,v_4)$



Graph Theory(Shortest Path)

تئوری گراف

```
procedure CUBE SHORTEST PATHS (A, C)
Step 1: for j = 0 to n - 1 do in parallel
            for k = 0 to n - 1 do in parallel
                        (1.1) if  j ≠ k and A(0, j, k) = 0         then
                                    A(0, j, k) ← ∞
                                end if
                        (1.2) B(0, j, k) ← A(0, j, k)
            end for
          end for
Step 2: for i = 1 to log(n - 1)l do
            (2.1) CUBE MATRIX MULTIPLICATION (A, B, C)
            (2.2) for j = 0 to n - 1 do in parallel
                        for k = 0 to n - 1 do in parallel
                                    (i) A(0, j, k) ← C(0, j, k)
                                    (ii) B(0, j, k) ← C(0, j, k)
                        end for
                    end for
            end for
```

$O(\log^2 n)$

Graph Theory(Shortest Path)

تئوری گراف

- ❖ Shortest Path
- ❖ هر کدام از مولفه های پردازنده دارای سه register به نام A، B و C است.
- ❖ A و B عناصر ماتریس و در C نتیجه قرار میگیرد.
- ❖ در cube multiplication به جای × و + از + و Min استفاده می شود.
- ❖ هر عنصر در ماتریس C کمترین وزن لازم جهت طی شدن مسیر از i به j است.
- ❖ (× ,+) => (+,Min)
- ❖ تحلیل:

- ➢ $t(n) = O(\log^2 n)$
- ➢ $p(n) = O(n^3)$
- ➢ $c(n) = O(n^3 \log^2 n)$

Graph Theory(Shortest Path)

تئوری گراف

	0	1	2	3	4	5	6
0	0	4	1	∞	7	∞	∞
1	∞	0	8	∞	∞	∞	∞
2	∞	∞	0	2	6	∞	∞
3	∞	5	∞	0	∞	∞	1
4	∞	∞	∞	∞	0	∞	∞
5	∞	∞	∞	2	1	0	∞
6	∞	3	∞	∞	∞	1	0

	0	1	2	3	4	5	6
0	0	4	1	3	7	∞	∞
1	∞	0	8	10	14	∞	∞
2	∞	7	0	2	6	∞	3
3	∞	4	13	0	∞	2	1
4	∞	∞	∞	∞	0	∞	∞
5	∞	7	∞	2	1	0	3
6	∞	3	11	3	2	1	0

	0	1	2	3	4	5	6
0	0	4	1	3	7	5	4
1	∞	0	8	10	14	12	11
2	∞	6	0	2	5	4	3
3	∞	4	12	0	3	2	1
4	∞	∞	∞	∞	0	∞	∞
5	∞	6	14	2	1	0	3
6	∞	3	11	3	2	1	0

	0	1	2	3	4	5	6
0	0	4	1	3	6	5	4
1	∞	0	8	10	13	12	11
2	∞	6	0	2	5	4	3
3	∞	4	12	0	3	2	1
4	∞	∞	∞	∞	0	∞	∞
5	∞	6	14	2	1	0	3
6	∞	3	11	3	2	1	0

Finding Roots Of Nonlinear Equations

ریشه یابی

- الگوریتم ترتیبی bisection method یکی از روشهای استاندارد برای یافتن ریشه است.
- اگر $f(x)$ تابع و a_0 و b_0 مقادیری باشند که $f(a_0)$ و $f(b_0)$ علامت متفاوتی دارند به طوری که: $f(a_0)\ f(b_0) < 0$
- ریشه $f(x)$ بین a_0 و b_0 است. نقطه میانی:

$$m_0=(a_0 + b_0)/2$$

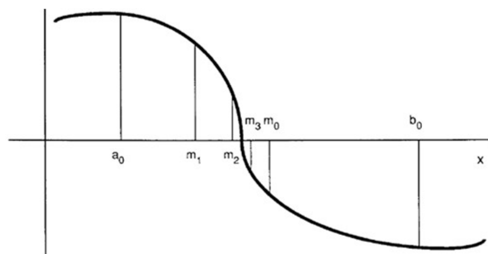
- اگر $f(a_0)\ f(m_0) < 0$ باشد ریشه بین a_0 و m_0 است و اگر $f(m_0)\ f(b_0) < 0$ باشد، ریشه بین m_0 و b_0 است. این عملیات آنقدر ادامه می یابد که
- $abs(f(m_0))<c'$ یا $abs(a_n,b_n)<c$
- که c' و c اعداد مثبت انتخاب شده ای هستند که دقت مطلوب را دارند.

Finding Roots Of Nonlinear Equations

ریشه یابی

```
procedure BISECTION (f, a, b, c)
while abs(b - a) ≥ c do
        (1) m ← (a + b)/2
        (2) if f (a) f (m) < 0 then b ← m
                         else a ← m
            end if
end while
```

- تحلیل:



- $t(n)=O(\log w)$
- $w=|b-a|$

Finding Roots Of Nonlinear Equations

ریشه یابی

❖ الگوریتم موازی BISECTION با N پردازنده روی CREW SM SIMD

❖ در این روش به جای N فاصله N+1 فاصله در نظر گرفته می شود.

❖ b-a به N قسمت تقسیم شده و N فاصله به N پردازنده داده می شود و همان الگوریتم قبل توسط هر پردازنده انجام می شود.

❖ اگر در این N قسمت ریشه پیدا نشد ریشه در قسمت N+1 است.

❖ تحلیل:

- $w/(N+1)^j < c$
- $t(n)=O(\log_{N+1} w)$
- $p(n)=O(N)$
- $c(n)=O(N \log_{N+1} w)$

Finding Roots Of Nonlinear Equations

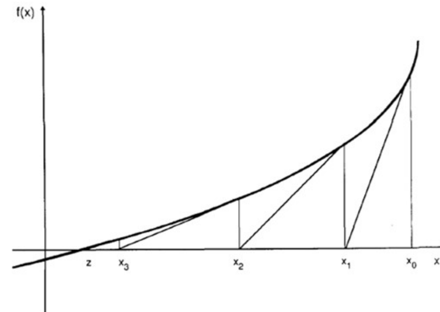
ریشه یابی

```
procedure SIMD ROOT SEARCH (f, a, b, c)
while (b - a) ≥ c do
        (1) s ← (b - a)/(N + 1)
        (2) y0 ← f (a)
        (3) yN+1 ← f (b)
        (4) for k = 1 to N do in parallel
                (4.1) yk ← f (a + ks)
                (4.2) if yk-1 yk < 0 then
                        (i) a ← a + (k - 1)s
                        (ii) b ← a + ks
                     end if
        end for
        (5) if yNyN+1 < 0 then
                a ← a + Ns
        end if
end while
```

Finding Roots Of Nonlinear Equations

ریشه یابی

- ❖ الگوریتم ترتیبی Newton
- ❖ در این روش برای محاسبه ریشه
- ❖ $x_{n+1}=x_n - f(x_n)/f'(x_n)$ for n = 0, 1, 2, ...
- ❖ تا $abs(x_{n+1}- x_n) < c$ و $f'(x_n)$ مشتق $f(x_n)$ است و c دقت مطلوب است.



Finding Roots Of Nonlinear Equations

ریشه یابی

- ❖ مانند الگوریتم قبل فاصله به N+1 فاصله تقسیم می شود.
- ❖ الگوریتم CRCW است.
- ❖ پردازنده ها همزمان شروع می کنند.
- ❖ ROOT ابتدا ∞ است و به محض آنکه مقدار آن تغییر کرد همه پردازش ها پایان می یابند.
- ❖ اگر دو پردازنده همزمان بخواهند مقدار ROOT را تغییر دهند پردازنده با شماره کوچکتر اجازه دسترسی دارد.
- ❖ R حداکثر تعداد تکرار است.

➢ $t(n)=O(\log m)$

- ❖ m تعداد مطلوب دقت ارقام بعد از اعشار برای پاسخ است.

Finding Roots Of Nonlinear Equations

ریشه یابی

```
procedure MIMD_ROOT_SEARCH (f, f', a, b, c, r, ROOT)
Step 1: s ← (b - a)/(N + 1)
Step 2: for k = 1 to N do
                create process k
        end for
Step 3: ROOT ← ∞
Step 4: Process k
        (4.1) x_old ← a + ks
        (4.2) iteration ← 0
        (4.3) while (iteration < r) and (ROOT = ∞) do
                (i) iteration ++
                (ii) x_new ← x_old - f (x_old)/f'(x_old)
                (iii) if abs(x_new - x_old) < c then
                        ROOT ← x_new
                   end if
                (iv) x_old ← x_new
          end while
```

Solving Linear Equation

حل معادلات خطی

- الگوریتم Gauss-Jordan
- n معادله n مجهول
- اگر n=4 باشد، برای حل آن x_1، x_2، x_3 و x_4 محاسبه می شوند:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = b_2$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = b_3$$
$$a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = b_4$$

- برای حل این معادلات عناصر قطر اصلی باقی مانده و بقیه از بین می روند.
- n^2+n پردازنده در یک $n \times n+1$ آرایه
- b_i در این روش $a_{i,n+1}$ در نظر گرفته شده است.

- $t(n)=O(n)$
- $p(n)=O(n^2)$
- $c(n)=O(n^3)$

Solving Linear Equation

حل معادلات خطی

```
procedure CREW SIMD GAUSS JORDAN (A, b, x)
Step 1: for j = 1 to n do
            for i = 1 to n do in parallel
                for k = j to n + 1 do in parallel
                    if (i # j) then
                        a_ik ← a_ik - (a_ij/a_jj)a_jk
                    end if
                end for
            end for
        end for
Step 2: for i = 1 to n do in parallel
            x_i ← a_i,n+1 /a_ii
        end for
```

- ❖ n+1 همان ستون b است.

Solving Linear Equation

حل معادلات خطی

- ❖ مثال:

$$2x_1+x_2=3$$
$$x_1+2x_2=4$$

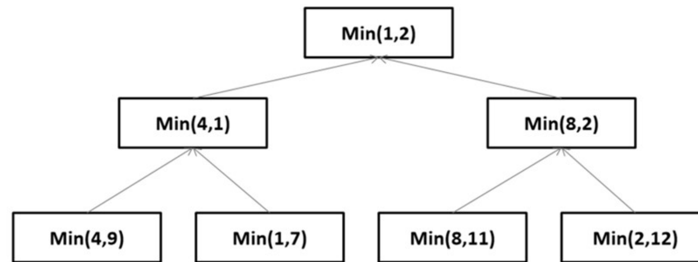
- Step 1:j=1
  - $a_{21}=a_{21}-(a_{21}/a_{11})a_{11}=1-(1/2)2=0$
  - $a_{22}=a_{22}-(a_{21}/a_{11})a_{12}=3/2$
  - $a_{23}=a_{23}-(a_{21}/a_{11})a_{13}=5/2$
- Step 1:j=2
  - $a_{12}=a_{12}-(a_{12}/a_{22})a_{22}=0$
  - $a_{13}=a_{13}-(a_{12}/a_{22})a_{23}=4/3$
- Step2: $x_i \leftarrow a_{i,n+1}/a_{ii}$
  - $x_1=2/3$ $x_2=5/3$

Minimum(tree)

پیدا کردن کوچکترین عدد

❖ یافتن کوچکترین عدد در مجموعه داده های {S={4,9,1,7,8,11,2,12

- N=n-1
- t(n) = O(log n)
- p(n) = O(n)
- c(n) = O(n log n)

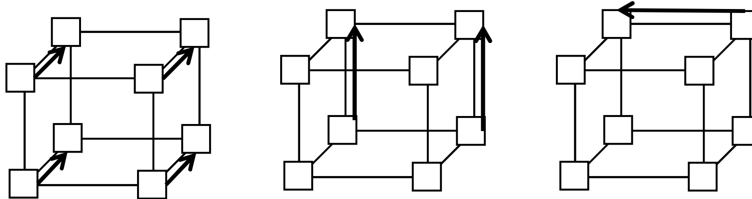


Maximum(Cube)

پیدا کردن بزرگترین عدد

❖ یافتن بزرگترین عدد در مجموعه

- t(n) = O(log N)
- p(n) = O(N)
- c(n) = O(N log N)



کلاس های پیچیدگی

❖ نظریه پیچیدگی شاخه ای از علوم کامپیوتر است که به سادگی یا سختی حل مسائل مختلف محاسباتی می پردازد.

❖ در تئوری پیچیدگی مسائل با توجه به زمان اجرا بر روی یک سیستم تک پردازنده (یا به طور دقیق تر یک دستگاه تورینگ قطعی)، به چندین کلاس پیچیدگی تقسیم می شود.

❖ مسائلی که زمان اجرای آنها با چند جمله ای در n محدود می شود متعلق به کلاس P (Polynomial) هستند. این مسائل که در زمان چندجمله ای حل می شوند را مسائل آسان می نامیم. حتی اگر چند جمله ای از درجه بالایی برخوردار باشد، به گونه ای که یک مساله بزرگ نیاز به چندین سال محاسبه بر روی سریعترین ابر رایانه موجود داشته باشد، هنوز امید وجود دارد که با بهبود الگوریتم یا عملکرد رایانه، زمان اجرای معقولی به دست آید.

کلاس های پیچیدگی

❖ مسائلی که دارای زمان اجرای نمایی هستند مسائل سخت نامیده می شوند. برای مثال، اگر مسئله ای با اندازه n به اجرای ۲^n دستورالعمل ماشین نیاز داشته باشد زمان اجرای آن برای n=100 روی پردازندهGIPS (giga IPS) حدود ۴۰۰ بیلیون قرن خواهد بود. زمانی که راه حل این گونه مسائل داده می شود در یک زمان چند جمله ای می توان صحت و درستی این راه حل را اثبات نمود. این مسئله متعلق به گروه NP (Non-deterministic Polynomial) است.

❖ یک نمونه از مسئله NP مسئله مجموع زیر مجموعه ها (Subset-sum) است. مجموعه ای از N عدد صحیح و جمع مورد نظر S در دست است. تعیین کنید که آیا زیر مجموعه ای از اعداد صحیح در مجموعه داده شده به S اضافه می شود یا خیر. مسئله ساده ای به نظر می آید ولی هنوز کسی حل آن را نمی داند و همه در تلاشند تا همه ۲^n زیر مجموعه را امتحان کنند. اگر هر یک از این آزمون ها وقت کمی لازم داشته باشند باز هم این مسئله برای n=100 قابل حل نیست. البته به این معنی نیست که نمی توانیم موارد خاصی از مسئله مجموع – زیر مجموعه ها را حل کنیم بلکه به این معنی است که الگوریتم کارا و مفید برای حل این مسئله هنوز مشخص نشده است.

کلاس های پیچیدگی

❖ این سوال که آیا P=NP یک مساله در تئوری پیچیدگی است.

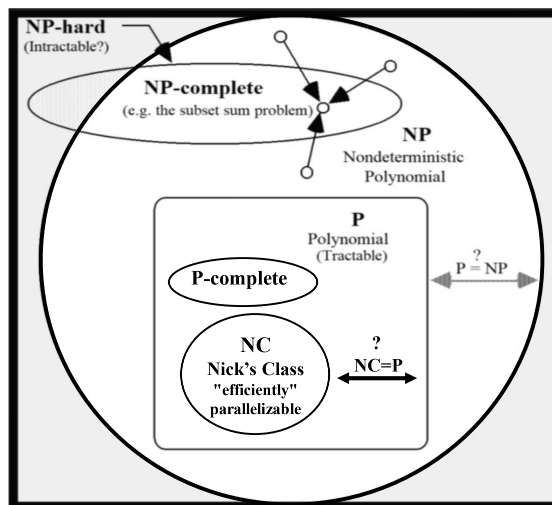
➢ پاسخ مثبت به این سوال بدان معنی است که مسئله مجموع زیر مجموعه ها و یک گروه از دیگر مسائل سخت(hard) را می توان به صورت کارا و مفید حل نمود حتی اگر هنوز هم هیچ راه حلی شناخته نشده باشد.

➢ پاسخ منفی نیز دلالت بر آن دارد که مسائلی وجود دارد که الگوریتم های کارا و مفید نیز نمی توانند آنها را حل کنند.

❖ محققان در تئوری پیچیدگی بر این باورند که P≠NP است و این مسائل را به عنوان مسائل NP-complete تعریف کرده اند. هر مسئله در NP را می توان با یک فرایند محاسباتی کارامد به هر یک از این مسئله ها تبدیل کرد. مسئله جمع زیر مجموعه ها خود یک مسئله NP-complete شناخته شده است بنابراین اگر فردی بهترین راه حل را برای آن پیدا نمود می تواند ثابت کند که P=NP است. از طرف دیگر اگر فردی اثبات کند که مسئله مجموع زیر مجموعه ها در P نیست آنگاه اثبات می شود که P≠NP است.

کلاس های پیچیدگی

❖ شکل زیر رابطه این کلاس ها را نشان می دهد.



کلاس های پیچیدگی

❖ اکثر مسائل در کلاس NP قرار دارند و زمان و منابع زیادی طی سالهای زیاد برای یافتن یک راه حل کارا برای این مسائل صرف شده است. با اثبات اینکه مسئله NP-complete است، هر گونه امیدی برای یافتن یک الگوریتم کارا برای حل مسئله را از بین می برد. بنابراین مسائل NP-complete سخت ترین مسائل در کلاس NP هستند. علاوه بر مسئله مجموع زیر مجموعه ها، مسائل زیر از لحاظ عملی از نوع NP-complete شناخته شده اند:

✓ ۱- اگر عبارت Boolean داشته باشیم که در آن متغیر ها مقدار صحیح بگیرند و به صورت ANDکردن چندین عبارت OR نوشته شده باشد که نتیجه عبارت TRUE باشد، تعیین مقدار متغیرها در این مسئله در کلاس NP قرار دارد حتی اگر هر عبارت OR محدود شده باشد که سه عملوند داشته باشد. (satisfiability)

✓ ۲- تعیین مقدار ۰ و ۱ به ورودی یک مدار منطقی که مقدار خروجی ۱ باشد.(circuit satisfiability)

✓ ۳- تصمیم در مورد اینکه گرافی دارای یک چرخه یا حلقه است که از تمام گره ها می گذرد مثل گراف همیلتونی (Hamiltonian cycle)

✓ ۴- پیدا کردن کوتاه ترین راه با کمترین هزینه در تعدادی از شهرها که باید هزینه سفر یا فاصله بین هر جفت شهر حداقل باشد. (مسئله فروشنده دوره گرد)

کلاس های پیچیدگی

❖ با توجه به دشواری مسئله NP، هنوز هم مسائل دیگری وجود دارد که در NP نیستند. این بدان معنی است که ادعای پاسخ به این مساله درست هست یا نه قابل حل نیست. مسئله NP-HARD از جمله مسائلی است که نمی دانیم واقعاً در NP هست یا نه اما می دانیم که مسئله NP را می توان با الگوریتم زمان چند جمله ای تبدیل نمود. نام این گروه حاکی از آن است که یک این چنین مسائلی حداقل به دشواری مسئله NP می باشند.

❖ برای اثبات این که مسئله ای مسئله NP-HARD است دارای دو بخش است:

✓ اثبات اینکه مسئله در NP هست با نشان دادن اینکه یک راه حل مشخص برای آن را می توان در زمان چند جمله ای اثبات کرد.

✓ با اثبات اینکه مسئله در NP-HARD است با نشان دادن اینکه بعضی مسائل Np-complete (و هر مسئله NP) را می توان به آن کاهش داد. طیف وسعی از مسائل NP-complete را داریم که می توانیم از آنها انتخاب کنیم.

کلاس های پیچیدگی

- پردازش موازی هیچ سودمندی برای حل مسائل NP ندارد. مسئله ای که ۴۰۰ بیلیون قرن حل آن در یک تک پردازنده طول می کشد، اگر قرار باشد که با بیش از یک بیلیون پردازنده انجام شود نیز ممکن است ۴۰۰ قرن به طول بکشد. این اظهارنظر به موارد خاصی اشاره ندارد بلکه به یک راه حل کلی برای همه موارد اشاره می کند. بنابراین پردازش موازی در ابتدا برای بالا بردن سرعت اجرای مسائل در P مفید می باشد. در اینجا حتی افزایش سرعت با ضریب ۱۰۰۰ به معنی تفاوت عملی بودن یا عملی نبودن است (اجرای چندین ساعت در مقابل یک سال)

- در سال ۱۹۷۹ Niclaus Pippenger اظهار داشت که مسائل قابل موازی سازی در P ممکن است به صورت مسائلی تعریف شوند که می توانند در یک بازه زمانی حل شوند اغلب این مسائل بصورت لگاریتمی هستند برای مثال $T(P)=O(\log^k n)$ برای مقادیر ثابت k، تعداد پردازنده بیشتری از یک چند جمله ای $p=O(n^l)$ استفاده نخواهد کرد. این گروه از مسائلی هستند که بعداً NC نامیده شدند و به صورت جامع و کامل مورد مطالعه و تحقیق قرار گرفتند و پایه ای برای نظریه پیچیدگی است.

کلاس های پیچیدگی

- Pippenger از ماشین های موازی معروف به parallel random-access machine (PRAM) برای فرموله کردن نتایج پیچیدگی استفاده کرد. یک شکل ضعیف تری از NC به صورت قضیه محاسباتی موازی شناخته شده است به شرح زیر بیان می شود.

- "هر چیزی که بتواند بر روی یک ماشین تورینگ با استفاده از یک فضای محدود چند جمله ای (polylogarithmically) در یک زمان نامحدود محاسبه شود می تواند بر روی یک ماشین موازی در زمان چند جمله ای (polylogarithmic) با استفاده از تعداد نامحدودی پردازنده محاسبه شود و بالعکس"

- مشکل این قضیه این است که هیچ حدی برای منابع محاسباتی غیر از زمان قرار نداده است. اهمیت NC و رواج آن ناشی از محدودیت منابع زمانی و سخت افزاری است، در حالی که در همان زمان نسبت به تفاوت های تکنولوژیکی و معماری در پیاده سازی ماشینهای موازی تقریبا بی تفاوت بودند.

کلاس های پیچیدگی

❖ در حال حاضر، سوال NC=?P یک مسئله باز از تئوری پیچیدگی می باشد. همانطور که در مورد سوال P=?NP هست. هیچ کس نمی داند که پاسخ این سوال چیست، اما یک گمان قوی وجود دارد که NC≠P. دلیل این گمان کاملا شبیه P≠NP می باشد. یک مسئله P-complete در P مسئله ای است که هر مسئله ای در P می تواند در زمان لگاریتمی بااستفاده از یک تعداد چند جمله ای پردازنده به آن مساله تبدیل شود. اگر یک الگوریتم با زمان لگاریتمی با یک تعداد چند جمله ای پردازنده برای هر مسئله در p-complete پیدا شود آنگاه تمام مسئله ها در P می توانند بطور کارا موازی شوند و آنگاه NC=P. برخی از این مسائل سالها است که وجود دارند و به طور جامع و کامل مورد تحقیق محققان متعددی قرار گرفته اند. بنابراین با فقدان یک الگوریتم کارامد برای چنین مسئله هایی به این نتیجه می رسیم که NC≠P.

❖ مرتب سازی یکی از بهترین مثالها برای مسئله NC می باشد. مسئله مقدار-مدار circuit-value یک مثال از یک مسئله P-complete می باشد که یک مدار منطقی با ورودی های مشخص ارائه شده است و فقط لازم است تا خروجی ها تعیین گردد.

Parallel programming with OpenMP

برنامه نویسی موازی

❖ تنظیمات در visual studio، C++ و از نوع console application

❖ در مسیر زیر:

➢ Properties-> configuration-> C/C++-> languages-> OpenMP support-> Yes

❖ اضافه کردن کتابخانه OpenMP

➢ #include <omp.h>

❖ هر کدی که در دستور زیر قرار بگیرد به تعداد core ها به صورت موازی اجرا می شود.

```
#pragma omp parallel
  {
   // Code inside this region runs in parallel
   cout <<"Hello!\n";
  }
```

Parallel programming with OpenMP

برنامه نویسی موازی

❖ در کد زیر حلقه تقسیم می شود بین core ها و هر core قسمت مربوط به خود را انجام می دهد و ممکن است ترتیب خروجی ها نامرتب باشد.

```
#pragma omp parallel
 #pragma omp for
 for( i=0; i<100; i++)
 {
  cout<<i<<"\t";
 }
 cout<<".\n";
```

❖ یک نوع خروجی

```
75  76  77  78  79  80  81  82  83  84
85  86  87  88  89  90  91  92  93  94
95  96  97  98  99  25  26  27  28  29
30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49
50  51  52  53  54  55  56  57  58  59
60  61  62  63  64  65  66  67  68  69
70  71  72  73  74  0   1   2   3   4
5   6   7   8   9   10  11  12  13  14
15  16  17  18  19  20  21  22  23  24
```

Parallel programming with OpenMP

برنامه نویسی موازی

❖ تاثیر موازی سازی در زمان اجرا

```
double startTime,endTime;
startTime=omp_get_wtime();
                زمان
 #pragma omp parallel private(i) or shared (i)
 #pragma omp for
 for( i=0; i<100000000; i++)
 {
  cout<<omp_get_num_threads()<<"\t";
  cout<<i<<"\t";   تعداد threadها
 }
 endTime=omp_get_wtime();
cout<<"\n Time elapsed:"<<endTime-startTime<<"\n";
```

Parallel programming with MPI

برنامه نویسی موازی

❖ تنظیمات در visual studio، C و از نوع console application

❖ نصب Microsoft Compute Cluster Pack SDK

❖ در مسیر زیر:

- Properties-> configuration->All configurations
- Properties-> configuration-> C/C++-> General-> Additional Include Directories-> C:\Program Files\Microsoft Compute Cluster Pack\Include
- Properties-> configuration-> C/C++-> Advanced->Compile As-> Compile as C Code (/TC)
- Properties-> configuration->Linker->Additional Library Directories-> C:\Program Files\Microsoft Compute Cluster Pack\Lib\i386
- Properties-> configuration->Linker->Input-> msmpi.lib

❖ اضافه کردن کتابخانه MPI

- #include "mpi.h"

Parallel programming with MPI

برنامه نویسی موازی

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

int main(int argc, char* argv[])
{
 printf ("Hello\n");
  return 0;
}
```

❖ برای اجرای این کد باید در command prompt

- …../mpiexec –n 4 nameOfFile

❖ خروجی

Hello
Hello
Hello
Hello

برنامه نویسی موازی

Parallel programming with MPI

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

int main(int argc, char* argv[])
{
 int  nTasks, rank ;
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&nTasks);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 printf ("Number of threads = %d, My rank = %d\n", nTasks, rank);
 MPI_Finalize();
 return 0;
}
```

❖ خروجی

```
Number of threads = 4, My rank = 2
Number of threads = 4, My rank = 0
Number of threads = 4, My rank = 1
Number of threads = 4, My rank = 3
```

برنامه نویسی موازی

Parallel programming with MATLAB

❖ در ابتدای کد موازی

➢ matlabpool('open',#);

❖ # تعداد core

❖ در انتهای کد موازی

➢ matlabpool('close');

❖ برای بدست آوردن تعداد mathlabpool ها

➢ matlabpool('size')

❖ برای اجرای موازی حلقه ها

```
parfor i = 1 : 100
    (code)
End
```

❖ نمی توان حلقه های موازی تو در تو داشت.

Parallel programming with MATLAB

برنامه نویسی موازی

```
matlabpool ('open',2);
x= zeros(10,10);
parfor i = 1:10
y= zeros(1,1);
    for j = 1:10
        y(j) = i;
    end
    y
    x(i,:) = y;
end
x
matlabpool close;
```

خروجی

```
Starting matlabpool using the parallel configuration 'local'.
Waiting for parallel job to start...
Connected to a matlabpool session with 2 labs.
x =
    1   1   1   1   1   1   1   1   1   1
    2   2   2   2   2   2   2   2   2   2
    3   3   3   3   3   3   3   3   3   3
    4   4   4   4   4   4   4   4   4   4
    5   5   5   5   5   5   5   5   5   5
    6   6   6   6   6   6   6   6   6   6
    7   7   7   7   7   7   7   7   7   7
    8   8   8   8   8   8   8   8   8   8
    9   9   9   9   9   9   9   9   9   9
   10  10  10  10  10  10  10  10  10  10
Sending a stop signal to all the labs...
Waiting for parallel job to finish...
y =    7   7   7   7   7   7   7   7   7   7
y =    6   6   6   6   6   6   6   6   6   6
y =    5   5   5   5   5   5   5   5   5   5
y =    4   4   4   4   4   4   4   4   4   4
y =    3   3   3   3   3   3   3   3   3   3
y =    2   2   2   2   2   2   2   2   2   2
y =    1   1   1   1   1   1   1   1   1   1
y =    9   9   9   9   9   9   9   9   9   9
y =   10  10  10  10  10  10  10  10  10  10
y =    8   8   8   8   8   8   8   8   8   8
Performing parallel job cleanup...
Done.
```

Parallel programming with MATLAB

برنامه نویسی موازی

```
tic
matlabpool ('open',2);
x= zeros(100,10);
parfor i = 1:100
y= zeros(1,10);
    for j = 1:10
        y(j) = i;
    end
    %y
    x(i,:) = y;
end
%x
matlabpool close;
toc
```

- ❖ در نرم افزار MATLAB موازی سازی روی داده های پیچیده کارایی دارد.
- ❖ در داده های کوچک و اعمال ساده for سریعتر از parfor انجام می شود.
- ❖ در این مثال for بسیار سریعتر از parfor انجام می شود.

Parallel programming with MATLAB

برنامه نویسی موازی

```
function parfortest()
tic;
N=1000;
for i=1:N
   sequential_answer=slow_fun(i);
end
sequential_time=toc

tic;
parfor i=1:N
  sequential_answer=slow_fun(i);
end
parallel_time=toc

matlabpool ('open',4);
tic;
parfor i=1:N
  sequential_answer=slow_fun(i);
end
parallelmatlabpool4_time=toc
matlabpool close;

matlabpool ('open',2);
tic;
parfor i=1:N
  sequential_answer=slow_fun(i);
end
parallelmatlabpool2_time=toc
matlabpool close;
end

function result=slow_fun(x)
   pause(0.001);
   result=x;
end
```

Parallel programming with MATLAB

برنامه نویسی موازی

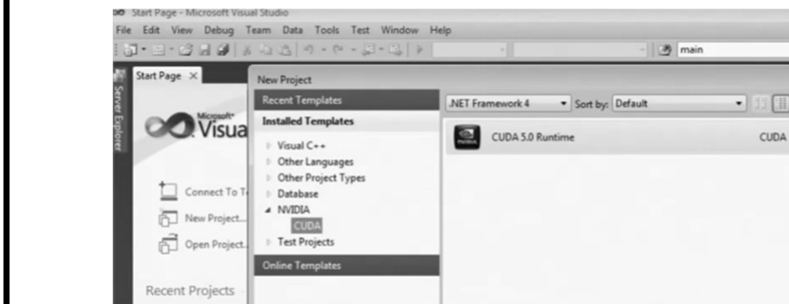
```
sequential_time =

  15.6020


parallel_time =

  15.6023

Starting matlabpool using the 'local' configuration ... connected to 4 labs.

parallelmatlabpool4_time =

   4.2054

Sending a stop signal to all the labs ... stopped.
Starting matlabpool using the 'local' configuration ... connected to 2 labs.

parallelmatlabpool2_time =

   7.8835

Sending a stop signal to all the labs ... stopped.
```

Parallel programming with CUDA(Visual studio C++ 2010)

برنامه نویسی موازی

❖ روش مناسبی جهت موازی کردن برنامه ها با استفاده از GPU

- https://developer.nvidia.com/cuda-zone
- https://developer.nvidia.com/cuda-toolkit
- Download and install a version of cuda
- NewProject->NVIDIA->CUDA



برنامه نویسی موازی

```
#include <stdio.h>
#define SIZE 1024
Void VectorAdd(int *a, int *b, int *c, int n)
{
    int i;
    for(i=0;i<n;i++)
        c[i]=a[i]+b[i];
}
int main()
{
    int *a,*b,*c;
    a=(int *) malloc(SIZE*sizeof(int));
    b=(int *) malloc(SIZE*sizeof(int));
    c=(int *) malloc(SIZE*sizeof(int));
    for(int i=0;i<SIZE;i++)
    {
        a[i]=i;
        b[i]=i;
        c[i]=0;
    }
    VectorAdd(a, b, c, SIZE);
    for(int i=0;i<SIZE;i++)
        printf("c[%d]=%d\n",i,c[i]);
    free(a);
    free(b);
    free(c);
    return 0;
}
```

۱) موازی سازی VectorAdd

۲) گرفتن فضا در GPU و ارسال داده ها به GPU

۳) تغییر دادن نحوه صدا زدن VectorAdd برای شروع اجرا در GPU

❖ در ابتدای کار داده ها باید از CPU درGPU کپی شوند.

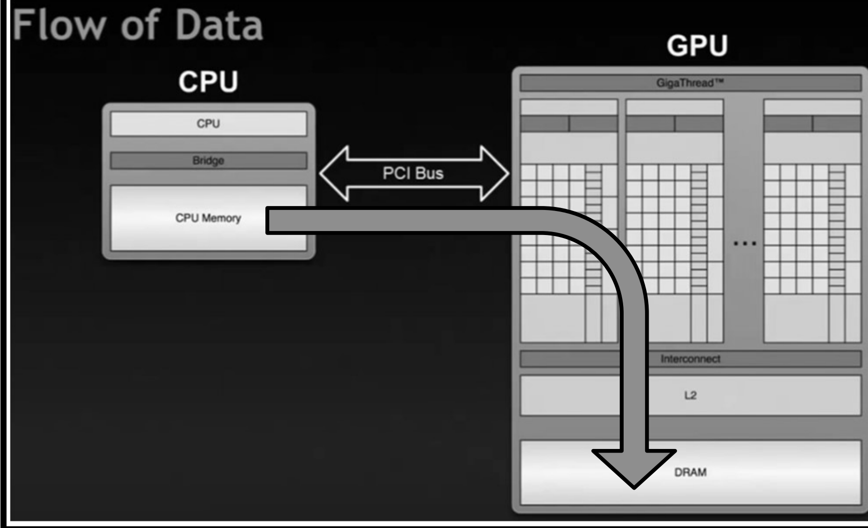
❖ عملیات در GPU انجام شود.

❖ نتیجه از GPU به CPU کپی شود.

Parallel programming with CUDA(Visual studio C++ 2010)

برنامه نویسی موازی

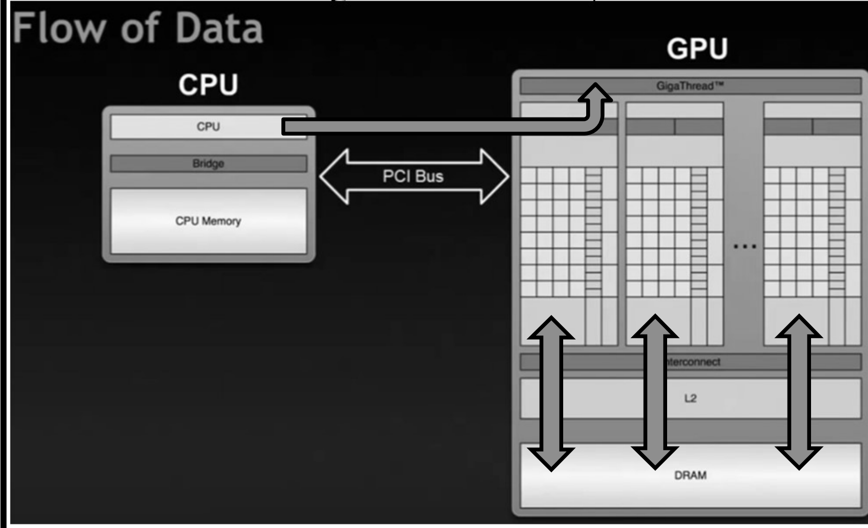
❖ در ابتدای کار داده ها باید از CPU درGPU کپی شوند.



Parallel programming with CUDA(Visual studio C++ 2010)

برنامه نویسی موازی

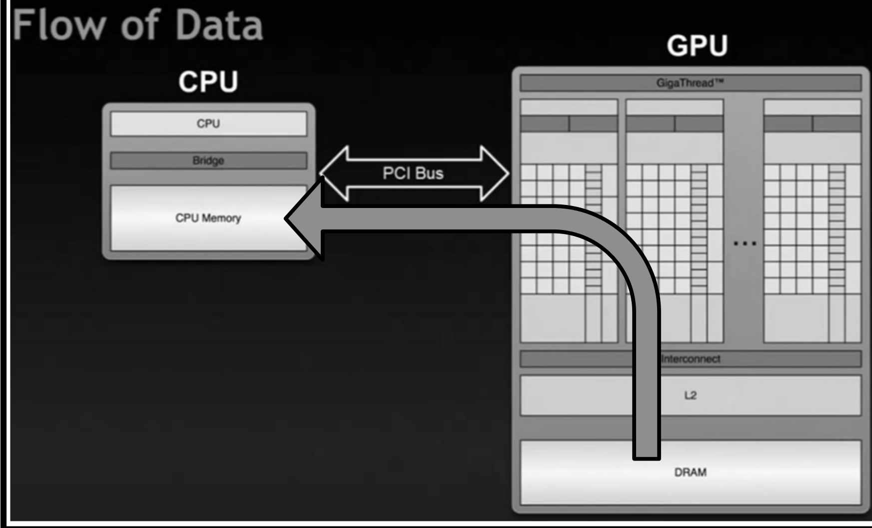
❖ عملیات در GPU انجام شود(VectorAdd شروع به کار می کند).



Parallel programming with CUDA(Visual studio C++ 2010)

برنامه نویسی موازی

❖ عملیات در GPU انجام شود(VectorAdd شروع به کار می کند).



Parallel programming with CUDA(Visual studio C++ 2010)

برنامه نویسی موازی

۱) موازی سازی VectorAdd

❖ روش مناسبی جهت موازی کردن برنامه ها با استفاده از GPU

➢ __global__

❖ به کامپایلر اعلام می کند این کد در GPU اجرا می شود.

➢ i=threadIdx.x

❖ یک thread روی یک عنصر کار می کند.

➢ if(i<n)

❖ نیازی به حلقه for نیست فقط باید چک شود که آیا داده دیگری هست یا خیر.

- Host CPU
- Device GPU

Parallel programming with CUDA(Visual studio C++ 2010)

برنامه نویسی موازی

۲) گرفتن فضا در GPU و ارسال داده ها به GPU

❖ حافظه گرفتن در GPU

- cudaMalloc(&d_a, SIZE*sizeof(int));

❖ کپی کردن داده ها در GPU

- cudaMemcpy(d_a,a,SIZE*sizeof(int), cudaMemcpyHostToDevice);

❖ کپی کردن داده ها در CPU بعد از انجام عملیات

- cudaMemcpy(c,d_c,SIZE*sizeof(int), cudaMemcpyDeviceToHost);

❖ آزاد کردن فضای حافظه در GPU

- cudaFree(d_a);

۳) تغییر دادن نحوه صدا زدن VectorAdd برای شروع اجرا در GPU

- VectorAdd<<<1,SIZE>>>(d_a, d_b, d_c, SIZE);

Parallel programming with CUDA(Visual studio C++ 2010)

برنامه نویسی موازی

```
#include <stdio.h>
#define SIZE 1024
__global__ Void VectorAdd(int *a, int *b, int *c, int n)
{
    int i=threadIdx.x;
    if(i<n)
        c[i]=a[i]+b[i];
}
int main()
{
    int *a,*b,*c;
    int *d_a,*d_b,*d_c;

    a=(int *) malloc(SIZE*sizeof(int));
    b=(int *) malloc(SIZE*sizeof(int));
    c=(int *) malloc(SIZE*sizeof(int));
    cudaMalloc(&d_a, SIZE*sizeof(int));
    cudaMalloc(&d_b, SIZE*sizeof(int));
    cudaMalloc(&d_c, SIZE*sizeof(int));

    for(int i=0;i<SIZE;i++)
    {
        a[i]=i;
        b[i]=i;
        c[i]=0;
    }

    cudaMemcpy(d_a,a,SIZE*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b,b,SIZE*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_c,c,SIZE*sizeof(int), cudaMemcpyHostToDevice);

    VectorAdd<<<1,SIZE>>>(d_a, d_b, d_c, SIZE);

    cudaMemcpy(c,d_c,SIZE*sizeof(int), cudaMemcpyDeviceToHost);

    for(int i=0;i<SIZE;i++)
        printf("c[%d]=%d\n",i,c[i]);

    free(a);
    free(b);
    free(c);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    return 0;
}
```

Parallel programming with CUDA(MATLAB)

برنامه نویسی موازی

- ابزارهای مختلف مانند GPUmat (رایگان) و Jacket (امکانات بیشتر)
- نصب cuda-toolkit
- دانلود GPUmat و خارج کردن از حالت zip
- پیدا کردن مسیر فایل GPUstart.m
- در MATLAB
- File-> Set Path -> Add Folder
- انتخاب و اضافه کردن پوشه GPUmat
- در command window محیط MATLAB، تایپ و اجرا GPUstart
- GPUstart: راه اندازی محیط GPU و بارگذاری کتابخانه های مورد نیاز
- GPUstop: توقف محیط استفاده از GPU
- GPUinfo: نمایش مشخصات مر بوط به CUDA و GPU

Parallel programming with CUDA(MATLAB)

برنامه نویسی موازی

- GPUsingle و GPUdouble معادل انواع single و double در MATLAB

تابع	توضیحات
A = GPUsingle(Ah) A = GPUdouble(Ah)	آرایه A را در GPU ایجاد می‌کند و آن را با استفاده آرایه Ah که در CPU قرار دارد مقداردهی می‌کند. این روش نیازمند انتقال اطلاعات از حافظه CPU به حافظه GPU است.
A = rand(size, GPUsingle) A = rand(size, GPUdouble)	آرایه A را در GPU ایجاد می‌کند و آن را با اعداد تصادفی مقداردهی می‌کند.
A = zeros(size, GPUsingle) A = zeros(size, GPUdouble)	آرایه A را در GPU ایجاد می‌کند و همه‌ی خانه‌های آن را با عدد صفر مقداردهی می‌کند.
A = ones(size, GPUsingle) A = ones(size, GPUdouble)	آرایه A را در GPU ایجاد می‌کند و همه‌ی خانه‌های آن را با عدد یک مقداردهی می‌کند.
A = colon(begin,stride, end, GPUsingle) A = colon(begin, stride,end, GPUdouble)	بردار A را در GPU ایجاد می‌کند و آن را با اعداد بین begin و end با گام stride پر می‌کند.

Parallel programming with CUDA(MATLAB)

برنامه نویسی موازی

- B=rand(10);
- A=GPUsingle(B); % A on GPU
- C=single(A)   % C on CPU

توابع پشتیبانی شده	مثال
عملگرهای پایه ریاضیات در MATLAB مانند: A*B, A-B, A.*B,A+B و غیره	A = rand(۱۰۰۰,GPUsingle); B = rand(۱۰۰۰,GPUsingle); C = A + B;
توابع عددی مانند: exp, sqrt, log و غیره	A = rand(1000,GPUsingle); B = rand(1000,GPUsingle); C = exp(A); D = sqrt(C) + B;
توابع فوریه	RE = rand(1000,GPUsingle); IM = i*rand(1000,GPUsingle); C = fft(RE + IM);

Parallel programming with CUDA(MATLAB)

برنامه نویسی موازی

❖ موازی سازی حلقه ها

```
GPUfor jt=1:10
%.....
GPUend
```

❖ i و j نمی توانند شمارنده حلقه باشند

```
GPUfor j=1:10
%.....
GPUend
```

✖

❖ حلقه خالی نمی توان ایجاد کرد

```
GPUfor jt=1:10
GPUend
```

✖

Parallel programming with CUDA(MATLAB)

برنامه نویسی موازی

❖ جمع دو ماتریس در MATLAB

```
A = single(rand(100)); % A is on CPU memory
B = single(rand(100)); % B is on CPU memory
C = A+B; % executed on CPU. C is on CPU memory
```

❖ جمع دو ماتریس در MATLAB با GPUmat

```
A = rand(100,GPUsingle); % A is on GPU memory
B = rand(100,GPUsingle); % B is on GPU memory
C = A+B; % executed on GPU. C is on GPU memory
```

❖ ارزیابی کارایی در MATLAB

```
A = rand(1000,1000); % A is on CPU
B = rand(1000,1000); % B is on CPU
tic;A.*B;toc;  % executed on CPU
```

```
A = rand(1000,1000,GPUsingle);
B = rand(1000,1000,GPUsingle);
tic;A.*B;GPUsync;toc;
```

منابع

منابع

References:

[1] S. G. Akl, ***The Design and Analysis of Parallel Algorithms***. USA: Prentice-Hall, 1989.
[2] H. El Rewini and M. Abd El Barr, ***Advanced Computer Architecture and Parallel Processing***. USA: John Wiley & Sons Publishing, 2005.
[3] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," ***IEEE Transactions on Computers***, vol. C-21, pp. 948-960, September 1972.
[4] W. F. McColl, "Special purpose parallel computing," in ***Lectures on parallel computation***, A. Gibbons and P. Spirakis, Eds., ed New York, USA: Cambridge University Press, 1993, pp. 261-336.
[5] B. Parhami, ***Introduction to Parallel Processing Algorithms and Architectures***. Santa Barbara, California: Kluwer Academic Publishers, 2002.
[6] A. Pimentel, ***Introduction to Parallel Architecture***: Universiteit van Amsterdam, 2002.
[7] G. Kotsis, "Interconnection Topologies and Routing for Parallel Processing Systems," Austrian Center for Parallel Computation, Austria1992.
[8] R. Miller and L. Boxer, ***Algorithms Sequential and Parallel: A Unified Approach***, Second Edition ed. Hingham, Massachusetts, 2005.
[9] K. A. Berman and J. L. Paul, ***Algorithms: sequential, parallel and distributed***. USA: Thomson, 2005.
[10] V. Kumar and V. N. Rao, "Parallel depth first search. Part II. analysis," ***Int. J. Parallel Program.***, vol. 16, pp. 501-519, 1987.
[11] L. R. Scott, ***et al.***, ***Scientific Parallel Computing***. New jersey, USA: Princeton University Press, 2005.
[12] Parallel Processing, Moldovan
[13] Masumeh Damrudi, PhD Thesis
[14] Syeda Mohsina Afroze No#990-00-4223 Advanced Logic Synthesis ECE 572