

الحواسيب التفرعية

Parallel Computers

- مقدمة حول الحساب التفرعي.
- الغاية من الحساب التفرعي.
- أنواع الحواسيب التفرعية:
 - الحاسب التقليدي.
 - النوع الأول: Shared Memory Multiprocessors
 - النوع الثاني: Message-passing Multicomputers
 - عناقيد الحواسيب المتناظرة: SMP Clusters
 - الحواسيب ذات الذاكرة المشتركة: Distributed Shared Memory
 - تصنيفات Flynn
- تقييم الأداء في الحساب التفرعي.

مقدمة: الحاجة إلى تسريع عمليات الحساب

Demand for Computational Speed

تعتبر الحاجة إلى تسريع عمليات الحساب حاجة دائمة ومتزايدة وهي تتجاوز في حجمها ما هو متاح حالياً. يظهر ذلك جلياً في مجالات، مثل:

□ مجالات النمذجة الرياضية Numerical modeling.

□ المحاكاة Simulation.

يجب التفكير في إنجاز الحسابات بأسرع زمن ممكن أو ضمن زمن معقول “reasonable time”.

مقدمة: مسائل التحديات الكبرى

Grand Challenge Problems

مسائل التحديات الكبرى هي المسائل التي لا يمكن حلها خلال فترات زمنية معقولة باستخدام الحواسيب المتوفرة في وقتنا الحالي.

هناك مسائل تحتاج لحلها إلى وقت قد يصل إلى حوالي ١٠ سنوات وهو وقت غير معقول بالنسبة لنا حالياً.

أمثلة:

□ نمذجة البنى الضخمة لـ DNA.

□ التنبؤ الجوي الشامل.

□ نمذجة حركة الأجرام السماوية.

التنبؤ الجوي

Weather Forecasting

يمكن نمذجة الغلاف الجوي بتقسيمه إلى خلايا ثلاثية الأبعاد وتجري الحسابات على مستوى كل خلية بشكل متكرر لمحاكاة عملية مرور الوقت.

مثال

إذا أخذنا الغلاف الجوي المحيط بالأرض حتى ارتفاع ١٠ ميل، فيمكن تقسيمه إلى خلايا من الأبعاد التالية: 1 mile X 1 mile X 1 mile. يبلغ عدد هذه الخلايا حوالي: 5×10^8 خلية.

إذا افترضنا أن كل عملية حساب تحتاج إلى ٢٠٠ عملية فاصلة عائمة floating point operations ليتم إنجازها، فمن أجل مرحلة واحدة من مراحل الحساب فإننا نحتاج إلى تنفيذ حوالي 10^{11} عملية فاصلة عائمة.

التنبوء الجوي

Weather Forecasting

فبالتالي من أجل التنبوء حول حالة الجو خلال فترة سبعة أيام القادمة على شكل فترات يفصل بينها مدة دقيقة واحدة، فإننا نحتاج إلى تنفيذ 10^{15} خطوة حساب (أي ما مجموعه عملية فاصلة عائمة). إذا استخدمنا حاسوب قادر على تنفيذ 100 Mflops، فإننا نحتاج لإنجاز عمليات الحساب إلى زمن وقدره 10^7 ثانية (أي إلى حوالي ١٠٠ يوم)!.
عملية فاصلة عائمة

لتنفيذ عمليات الحساب خلال فترة ١٠ دقائق، فإننا نحتاج إلى حاسوب قادر على تنفيذ حوالي 1.7×10^{12} عملية فاصلة عائمة في الثانية الواحدة).
1.7 Tflops (أي حوالي

نمذجة حركة الأجرام السماوية

Modeling Motion of Astronomical Bodies

يقوم كل جرم بجذب بقية الأجرام السماوية الأخرى إليه بسبب قوى الجاذبية
Gravitational Forces الكامنة فيما بينها.

إذا كان لدينا N جرم، فإننا نحتاج إلى حساب $N-1$ قوة من أجل كل جرم. أي نحتاج
إلى تنفيذ حوالي N^2 عملية حساب من أجل كافة الأجرام (نحتاج إلى تنفيذ $N \log_2 N$
عملية باستخدام خوارزمية حساب فعالة (Efficient Algorithm)).

فإذا كان لدينا مجرة مؤلفة من 10^{11} نجم، وكان بالإمكان تنفيذ كل عملية من عمليات
الحساب بزمان وقدره حوالي $1 \mu s$ (وهو زمن صغير جداً) فإننا نحتاج إلى 10^9
سنة لتنفيذ تكرار واحد من الحسابات في حال استخدامنا للخوارزمية التي تقتضي
تنفيذ N^2 عملية حساب. ونحتاج إلى زمن وقدره سنة واحدة في حال استخدامنا
للخوارزمية التي تقتضي تنفيذ $N \log_2 N$ عملية حساب.

الغاية من الحساب التفرعي

نحتاج إلى استخدام أكثر من حاسوب أو حاسوب مزود بأكثر من معالج لإيجاد حلول لمسائل ذات كميات حساب ضخمة. وذلك من أجل:

□ تنفيذ عمليات الحساب بشكل أسرع على الرغم من أن استخدامنا لـ n معالج لا يعني أننا سنحصل على النتيجة بزمن أقل بـ n مرة من قبل (هذه فكرة خاطئة لعدة أسباب سنذكرها فيما بعد).

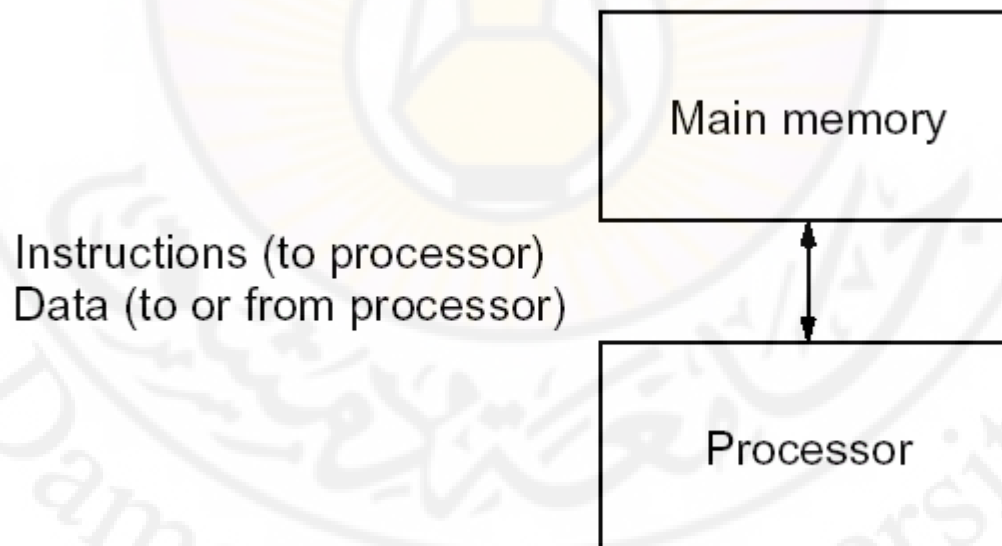
□ التسامح مع الأخطاء Fault Tolerance التي قد تقع، إضافة إلى توفير ذاكرة أكبر وغير ذلك من مصادر الحاسوب...

ظهر مفهوم الحواسيب التفرعية **Parallel Computers** والبرمجة التفرعية **Parallel Programming** المرتبط بها منذ أكثر من ٤٠ عاماً.

الحاسوب التقليدي

Conventional Computer

- يتألف الحاسوب التقليدي من معالج لتنفيذ البرامج المخزنة ضمن الذاكرة الرئيسية.
- يجري الوصول إلى مواضع الذاكرة الرئيسية بواسطة عناوينها التي تبدأ من العنوان 0 إلى العنوان $2^n - 1$ إذا كان لدينا n خانة للعنونة.



أنواع الحواسيب التفرعية

Types of Parallel Computers

يتوفر لدينا نوعين من الحواسيب التفرعية، وهي:

□ متعددة المعالجات ذات الذاكرة المشتركة Shared memory multiprocessors

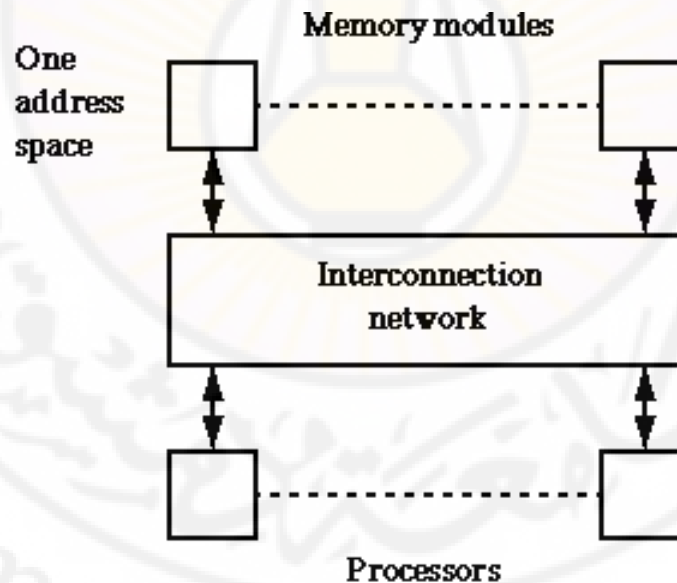


Figure 1.2 Shared memory multiprocessor model

أنواع الحواسيب التفرعية

Types of Parallel Computers

متعددات المعالجات ذات الذاكرة الموزعة Distributed memory multiprocessors. □

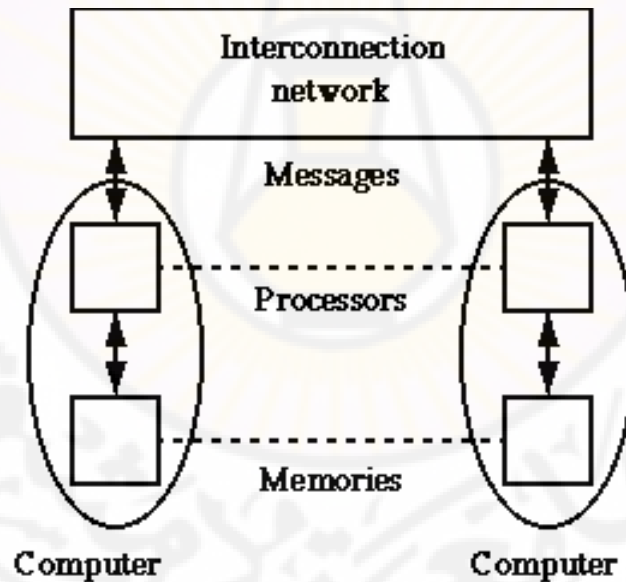


Figure 1.3 Message-passing multiprocessor model (multicomputer)

تصنيفات Flynn

Flynn's Classifications

قام Flynn في عام ١٩٦٦ بتصنيف الحواسيب اعتماداً على طريقة تعاملها مع المعطيات والتعليمات وفقاً لما يلي:

- ❑ **Single Instruction Stream-Single Data Stream (SISD) Computer**

- ❑ **Multiple Instruction Stream-Multiple Data Stream (MIMD) Computer**

يقوم كل حاسوب بتنفيذ برنامج مستقل ويتعامل هذا البرنامج مع معطيات خاصة به. يعتبر نموذجاً الحواسيب متعددة المعالجات ذات الذاكرة المشتركة والمعتمدة على تبادل الرسائل مثالين من أمثلة البنية MIMD.

- ❑ **Single Instruction Stream-Multiple Data Stream (SIMD) Computer**

أنواع الحواسيب التفرعية

Types of Parallel Computers



Figure 1.4 MIMD model

أنواع الحواسيب التفرعية

Types of Parallel Computers

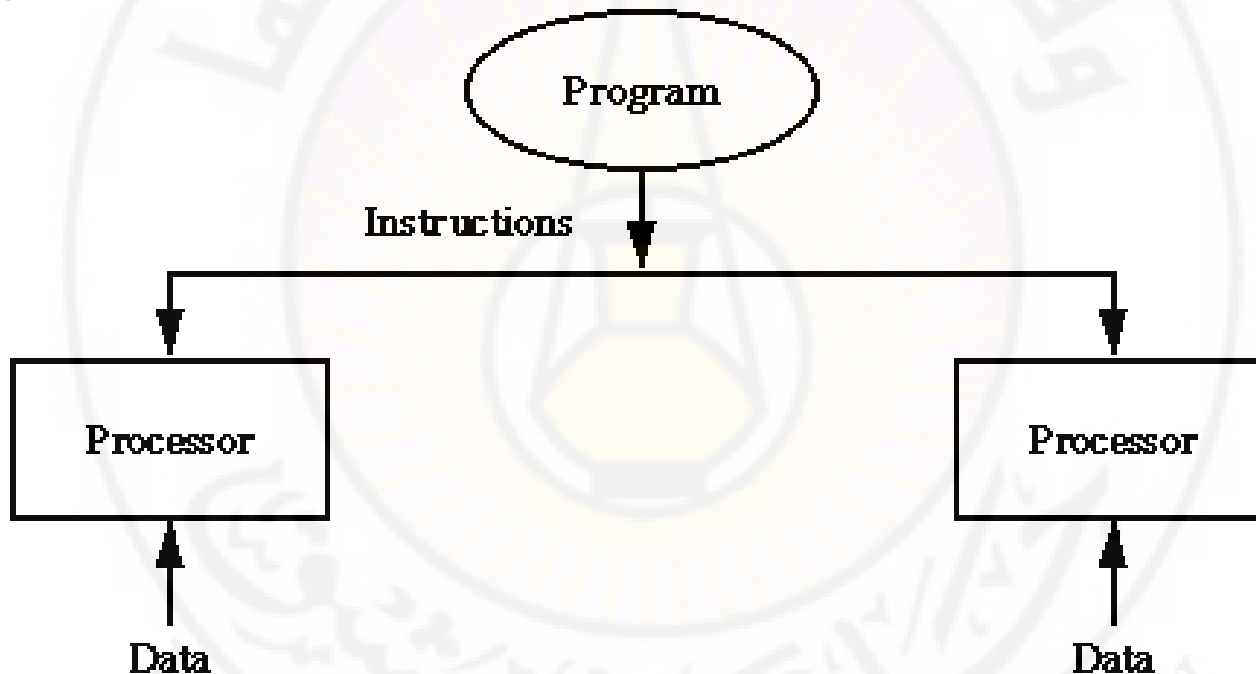
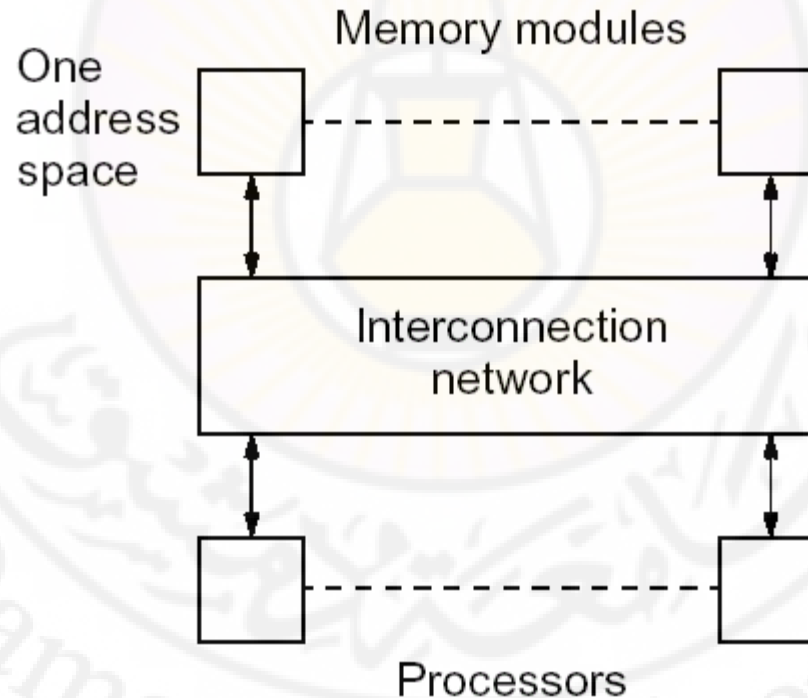


Figure 1.5 SIMD model

الحواسيب متعددة المعالجات ذات الذاكرة المشتركة

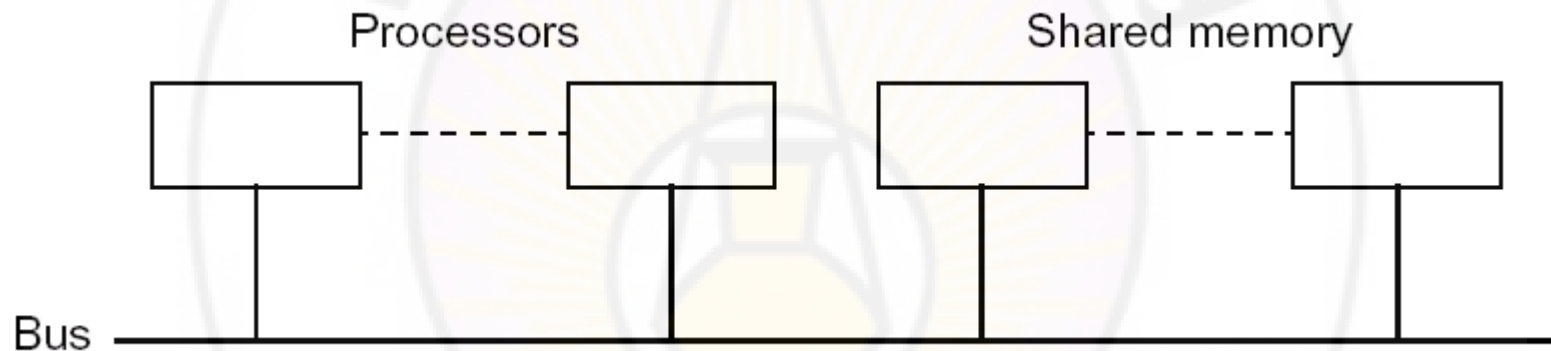
Shared Memory Multiprocessor

تعتبر هذه الحواسيب التوسع الطبيعي للحواسيب التقليدية. وهي تتألف من عدة معالجات قادرة على الوصول إلى ذاكرة مشتركة فيما بينها.



نظرة تبسيطية للحواسيب متعددة المعالجات ذات الذاكرة المشتركة

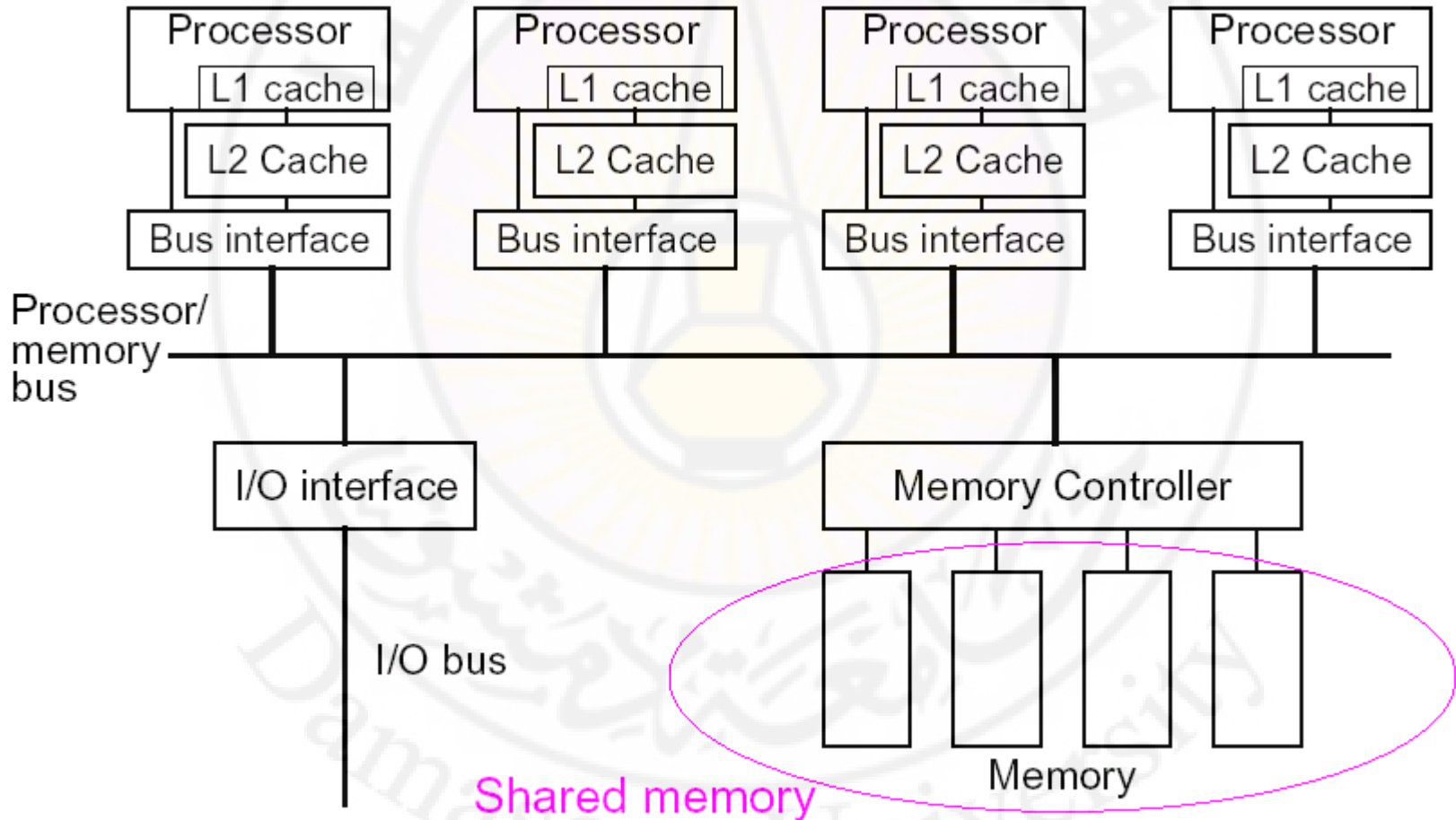
Simplistic view of a small shared memory multiprocessor



أمثلة:

- Dual Pentiums
- Quad Pentiums

Quad Pentium Shared Memory Multiprocessor



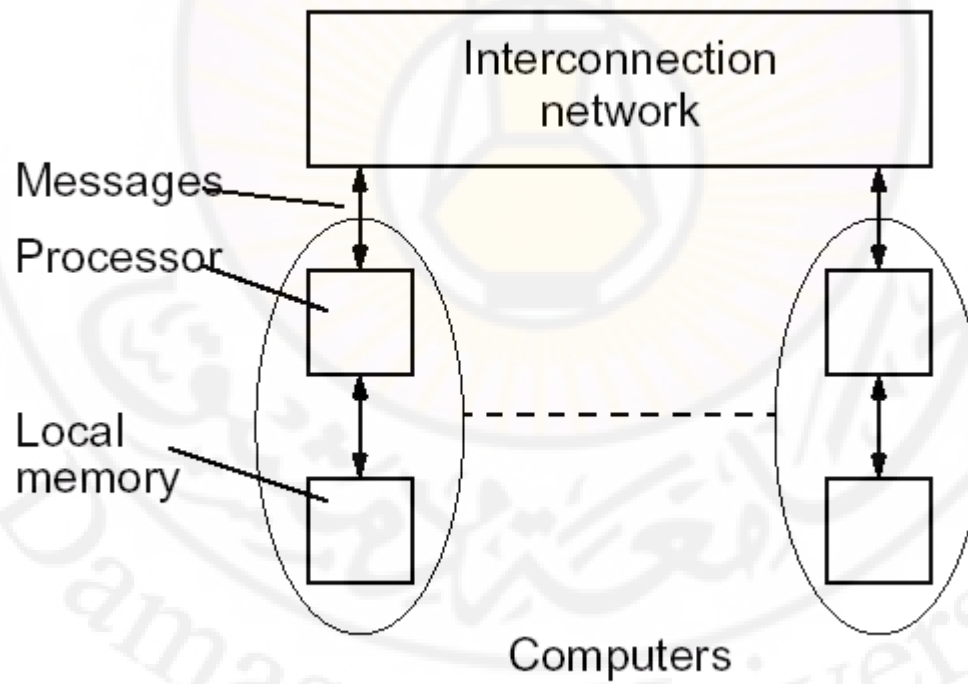
الحواسيب متعددة المعالجات ذات الذاكرة المشتركة

Shared Memory Multiprocessor

- يمكن لأي معالج أن يصل إلى أي عنوان من عناوين الذاكرة الرئيسية المشتركة.
- يتوفر لكافة المعالجات فضاء ذاكرة وحيد Single address space.
- يفضل المبرمجون التعامل معها رغم أنها تفرض عليهم توفير أساليب لحماية المعطيات المشتركة وتنظيم عملية الوصول إليها (باستخدام المقاطع الحرجة Critical sections).
- لبرمجة هذا النوع من الحواسيب، يوجد عدة طرق نذكر منها: المسالك Threads، المكتبات على مستوى المستثمر user-level libraries، لغات البرمجة التفرعية Parallel programming languages والمترجمات التفرعية Parallelizing compilers.

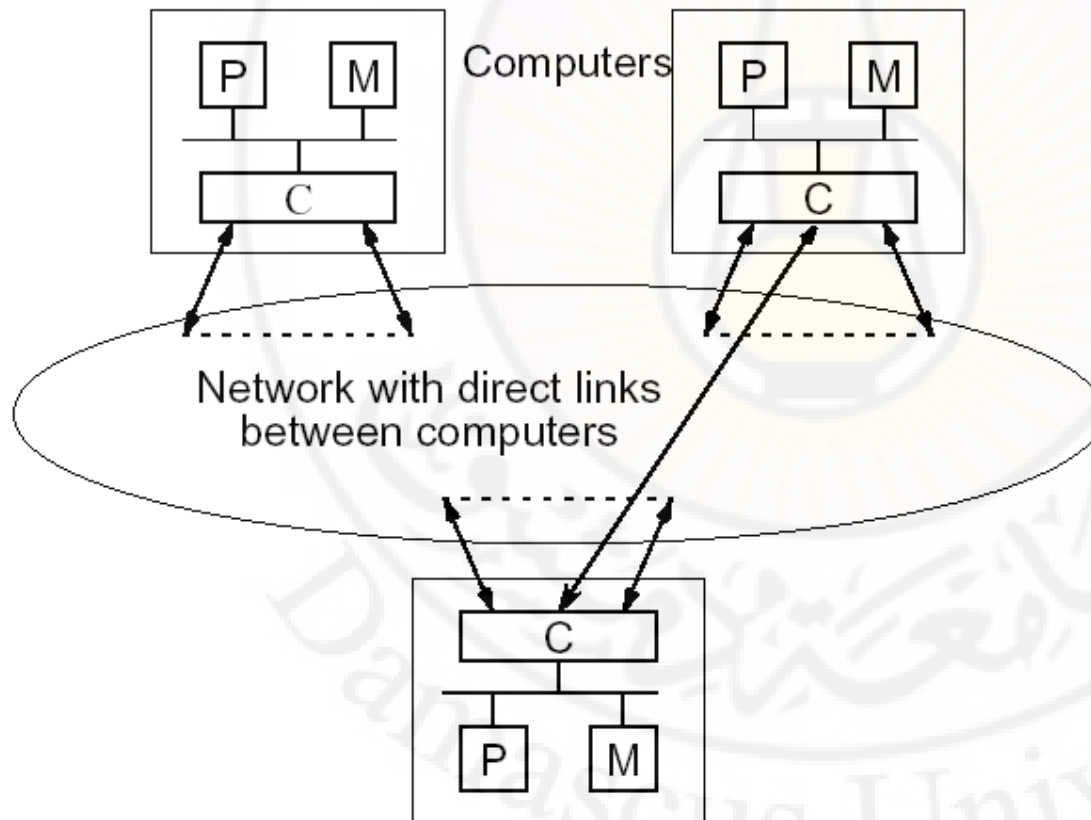
متعددات المعالجات الممتدة على تمرير الرسائل Message-Passing Multicomputer

وهي مجموعة من الحواسيب المرتبطة ربطاً تاماً فيما بينها بواسطة شبكة للوصل.



متعددات المعالجات ذات الشبكة الثابتة والمعدة على تمرير الرسائل

Static Network Message-Passing Multicomputer



تتصل الحواسيب فيما بينها
بواسطة خطوط اتصال
مباشرة.

شبكات الوصل الثابتة

Static Link Interconnection Networks

يوجد منها عدة أنواع:

- الشبكات الحلقية Ring.
- الشبكات الشجرية Tree.
- الشبكات المصفوفية ثنائية وثلاثية الأبعاد 2-D and 3-D arrays.
- الشبكات المكعبة الفائقة Hypercube.

الشبكات الشجرية Tree

Tree Structure

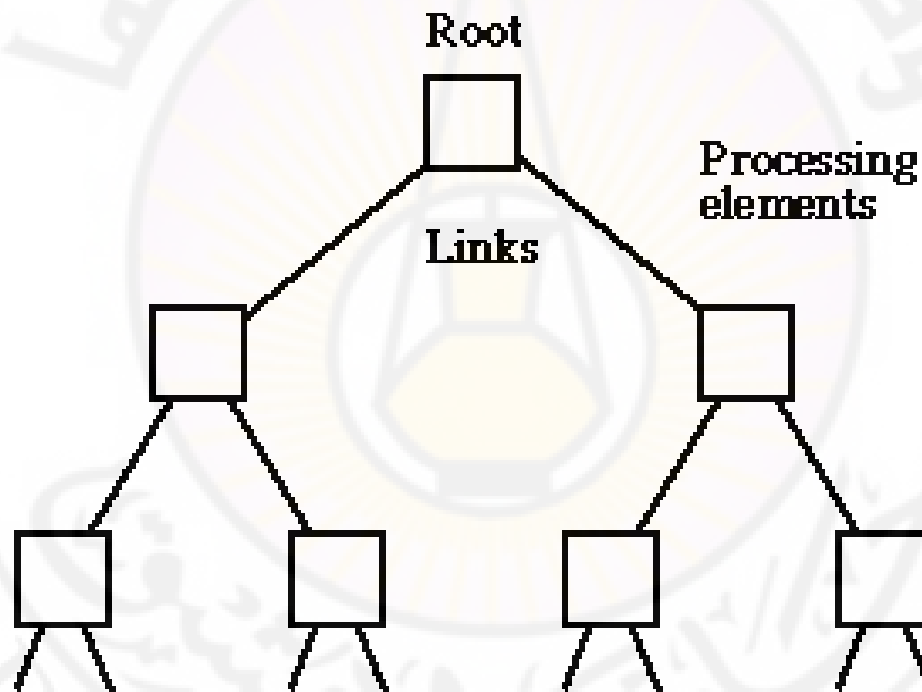
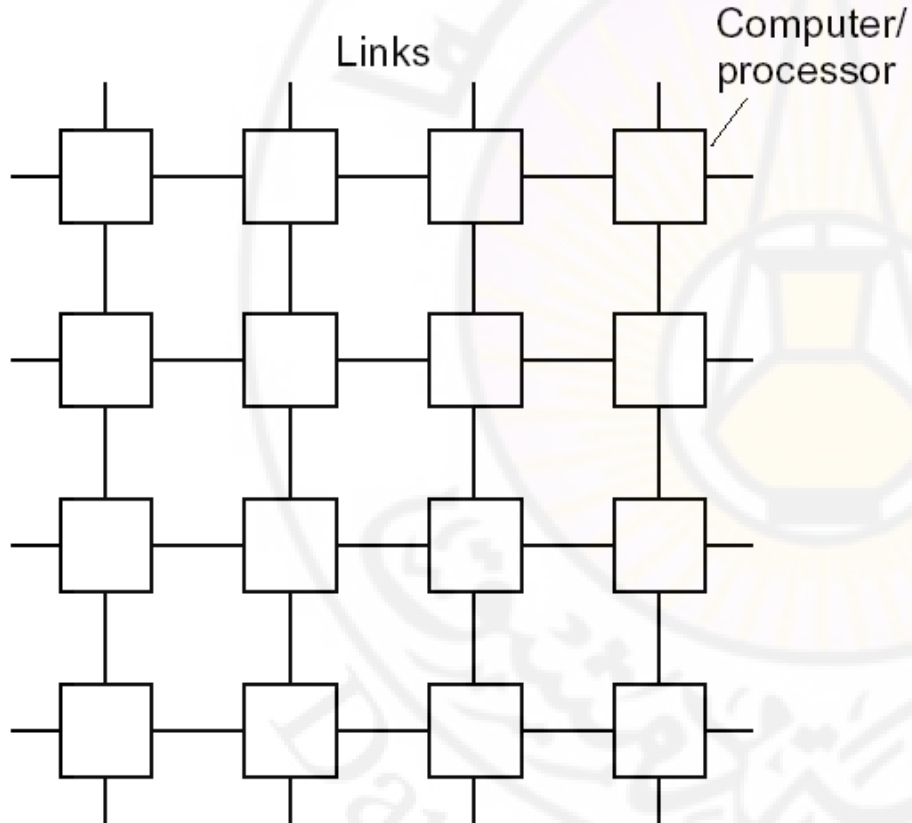


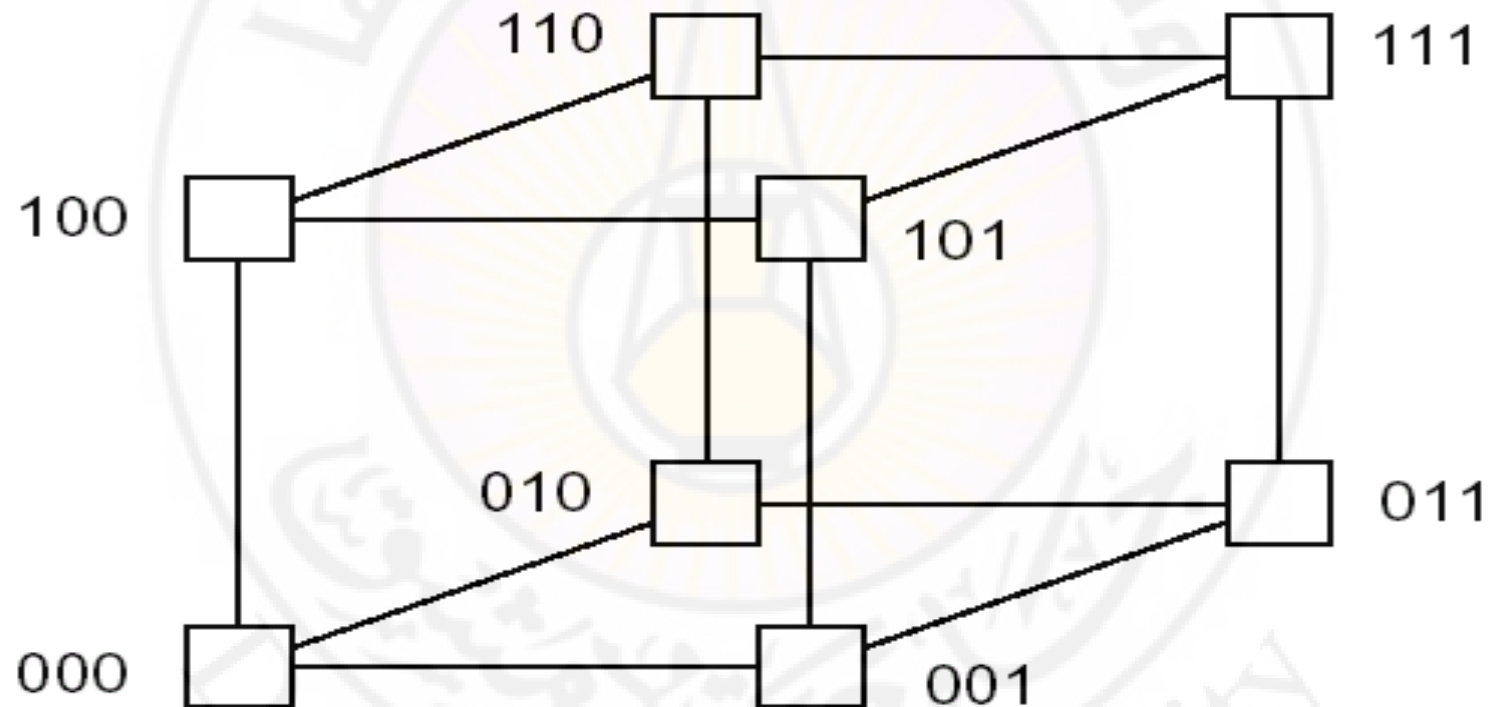
Figure 1.9 Tree structure

Two-dimensional array (mesh)

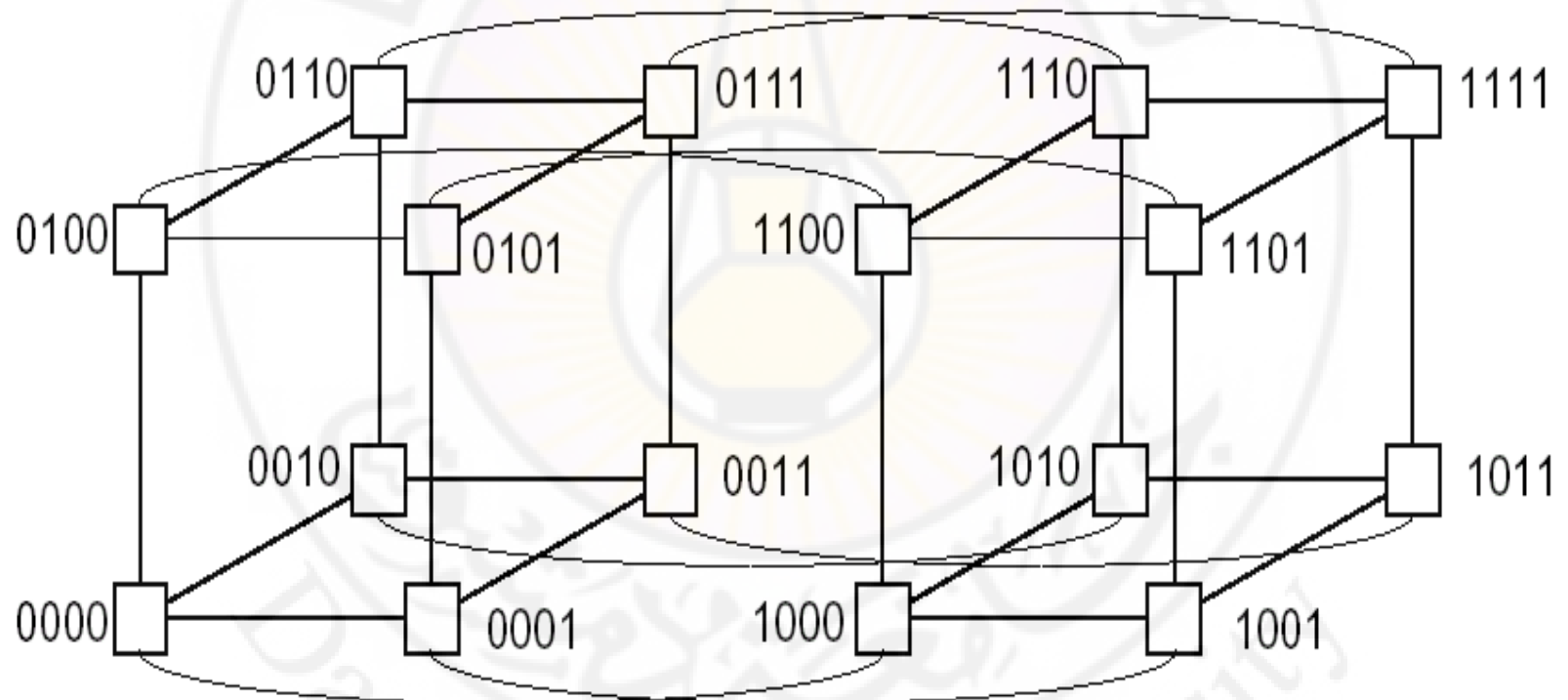


يمكن أيضاً استخدام شبكة مصفوفية
ثلاثية الأبعاد لتشكيل أنظمة ذات
أداء عالي.

Three-dimensional hypercube



Four-dimensional hypercube



Folded Torus

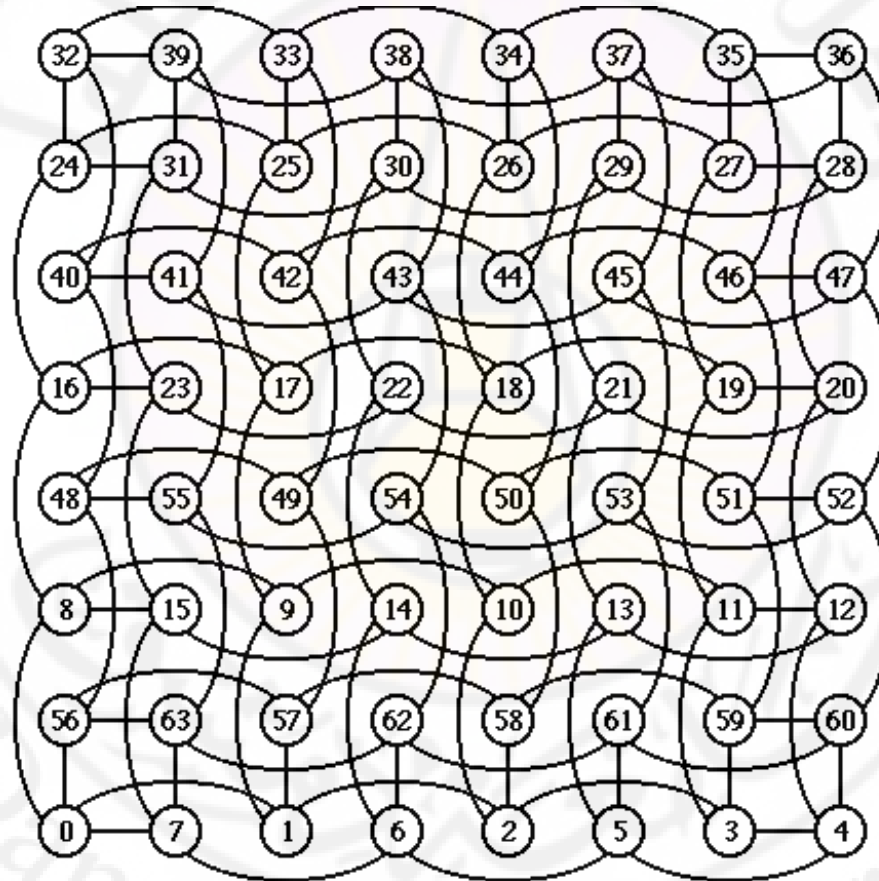


Figure 1.8 Folded torus

تغطية بنية ضمن أخرى

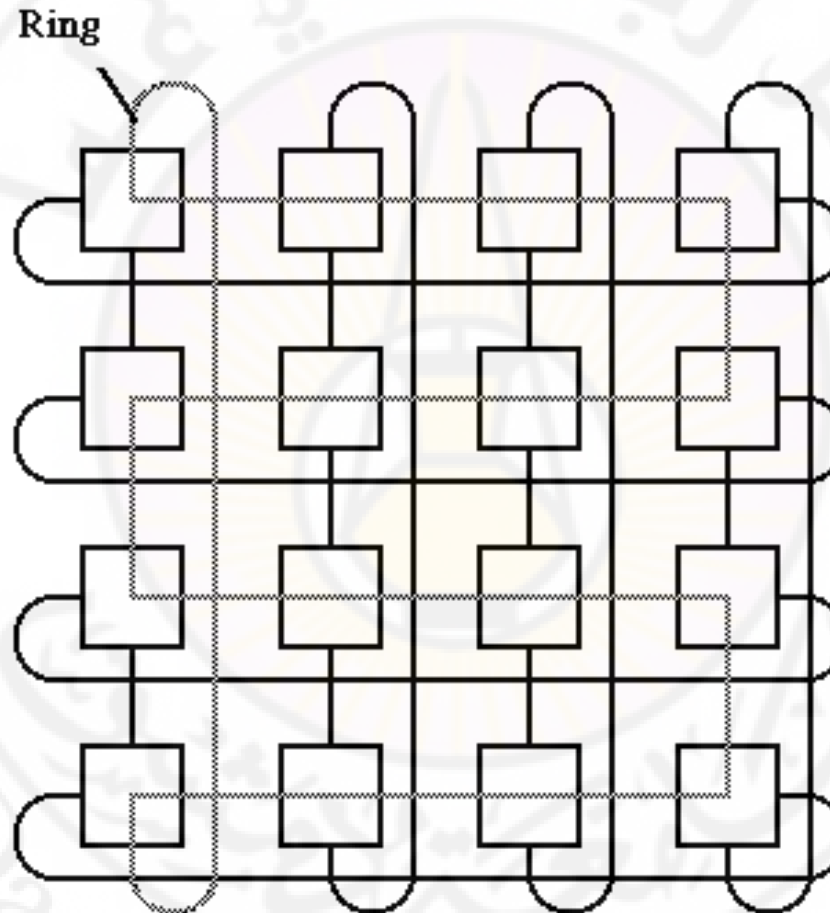


Figure 1.13 Embedding a ring into a torus

الحواسيب المعتمدة على شبكات الاتصال

Networked Computers as a Multicomputer Platform

تعتبر شبكة محطات العمل (Network of Workstations (NOW واحدة من الخيارات البديلة غير المكلفة مقارنة بالحواسيب الضخمة ذات الأداء العالي.

بدء هذا النوع من الحواسيب بالظهور منذ عام ١٩٩٠ من خلال عدد من المشاريع التي نذكر منها مشروع Berkeley NOW Project.

تتميز هذه الحواسيب بما يلي:

- تضم حواسيب شخصية ومحطات عمل ذات أداء عالي ورخصة الثمن.
- يمكن مكاملة المعالجات الحديثة بكل سهولة حال توفرها.
- يمكن استخدام البرمجيات المتوفرة معها وتعديلها بسهولة.

مثال: عناقيد بيوولف Beowulf Clusters.

الأدوات البرمجية لعناقيد الحواسيب

Software Tools for Clusters

لبرمجة عناقيد الحواسيب، يمكن أن نستخدم واحدة من الأدوات البرمجة التالية:

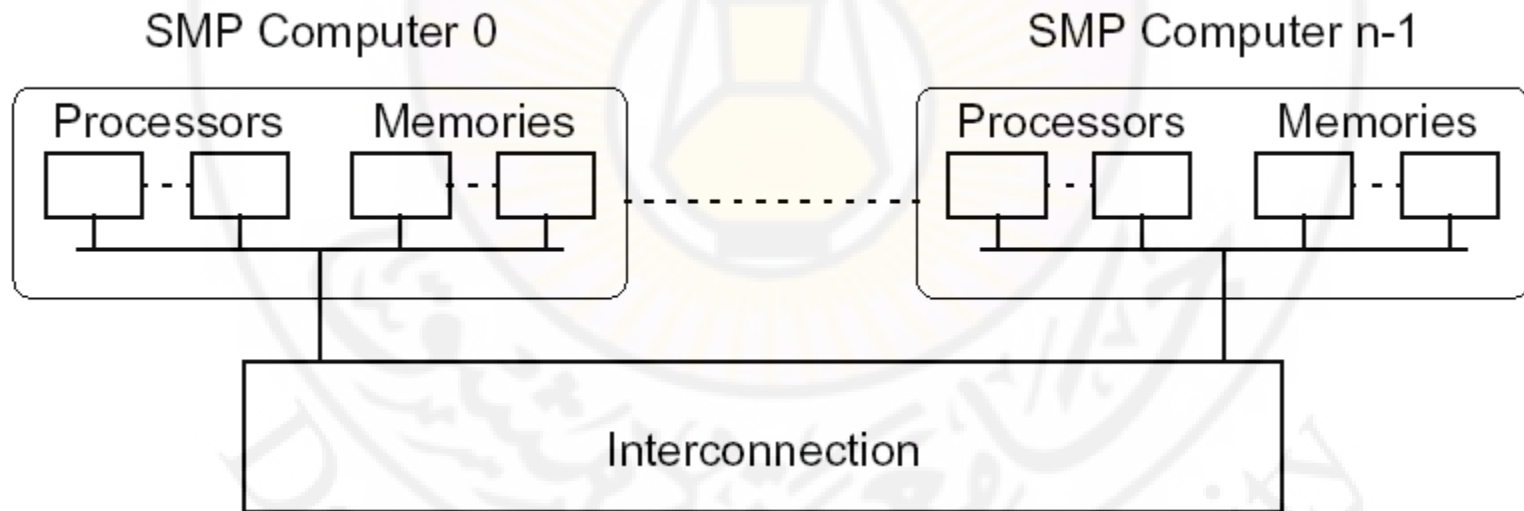
□ Parallel Virtual Machine (PVM): التي جرى تطويرها في نهاية الثمانينات من القرن الماضي وأصبحت بعد ذلك مشهورة جداً.

□ Message Passing Interface (MPI) وهو معيار لتبادل الرسائل جرى تعريفه في التسعينيات من القرن الماضي.

توفر كلا الأدوات المذكورتين أعلاه مكتبات على مستوى المستخدم لتبادل الرسائل ويمكن استخدامهما مع لغات البرمجة C, C++ وغيرها.

SMP Cluster

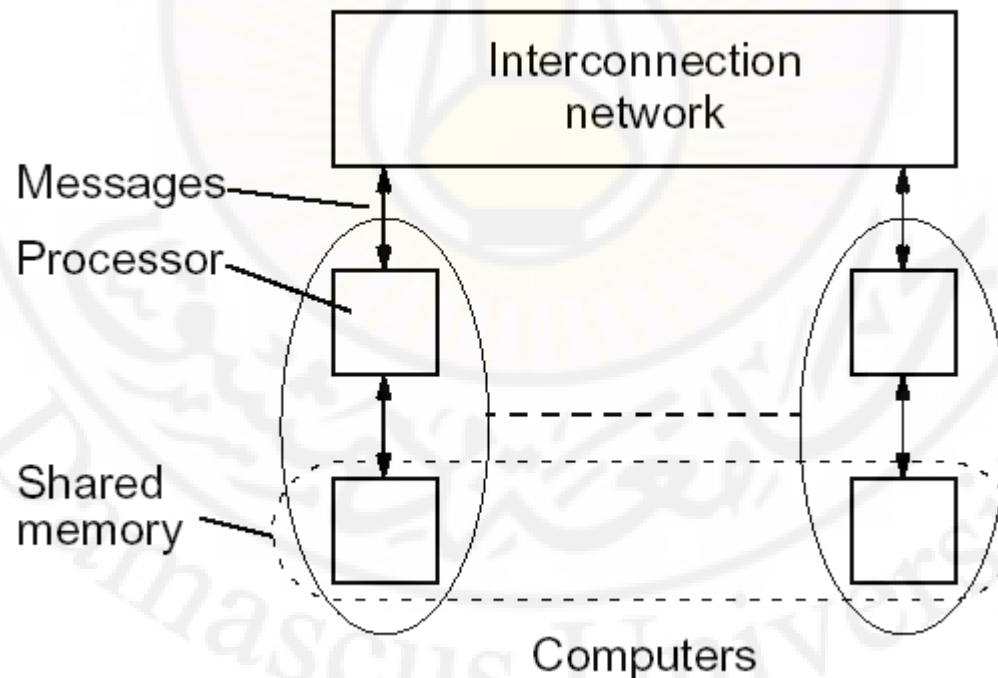
يمكن أن نستخدم أيضاً عناقيد من حواسيب ذات ذواكر مشتركة.



Distributed Shared Memory

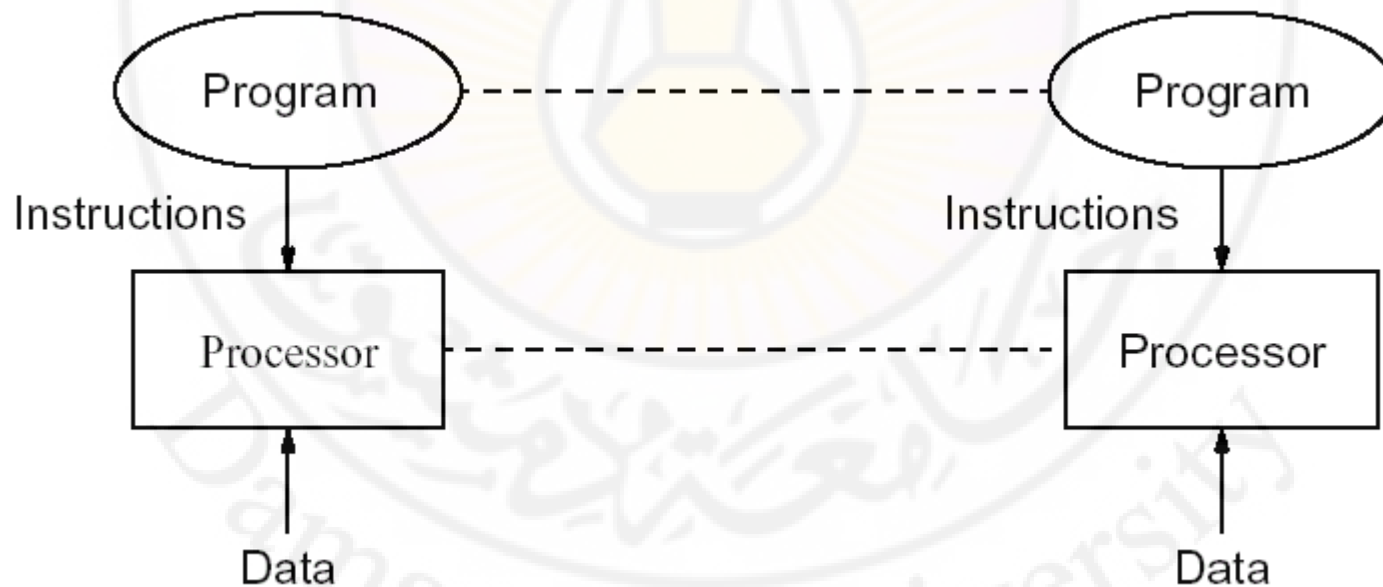
يمكن تحقيق ذلك بأن نجعل الذاكرة الموزعة لعنقود الحواسيب تبدو كما لو أنها تشكل فضاء عناوين وحيد لذاكرة واحدة.

يمكن في هذا النوع من الحواسيب استخدام أساليب برمجة الحواسيب ذات الذاكر المشتركة.



Multiple Program Multiple Data (MPMD) Structure

تعتبر هذه البنية جزءاً مرتبطاً بالبنية من النمط MIMD حيث يمتلك كل معالج برنامجاً خاصاً به.



Single Program Multiple Data (SPMD) Structure

يقوم كل معالج بتنفيذ نفس البرنامج ولكن بالتعامل مع معطيات مختلفة وبشكل مستقل عن المعالجات الأخرى.

يمكن أن يكون البرنامج المنفذ مهيكلًا بطريقة تسمح لبعض المعالجات بتنفيذ أجزاء محددة دون أن تستطيع المعالجات الأخرى فعل ذلك.

معامل الاستجابة

Speedup Factor

$$S(p) = \frac{\text{Execution time using one processor (single processor system)}}{\text{Execution time using a multiprocessor with } p \text{ processors}} = \frac{t_s}{t_p}$$

تزداد قيمة معامل الاستجابة باستخدام الحواسيب متعددة المعالجات. كما يرتبط ذلك ببنية الخوارزمية التفرعية المستخدمة.

يمكن أيضاً التعبير عن معامل الاستجابة باستخدام عدد خطوات الحساب كما يلي:

$$S(p) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with } p \text{ processors}}$$

تعتبر القيمة p عن قيمة الاستجابة العظمى في حال استخدام p معالج (وهذا ما يسمى بالاستجابة العظمى).

$$S(n) = \frac{t_s}{t_s/p} = p$$

الفعالية Efficiency

$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}}$$
$$= \frac{t_s}{t_p \times p}$$

which leads to

$$E = \frac{S(p)}{p} \times 100\%$$

وهي تحدد نسبة الزمن الذي جرى ضمنه استخدام المعالجات لتنفيذ أعمال حسابية.

نحصل على فعالية 100% عندما تعمل كافة المعالجات كامل الوقت وتحقق معامل استجابة مساوي للقيمة p .

الكلفة (العمل) Cost (Work)

Cost = Execution time $\times$ number of used processors

كلفة البرنامج التسلسلي t_s فقط

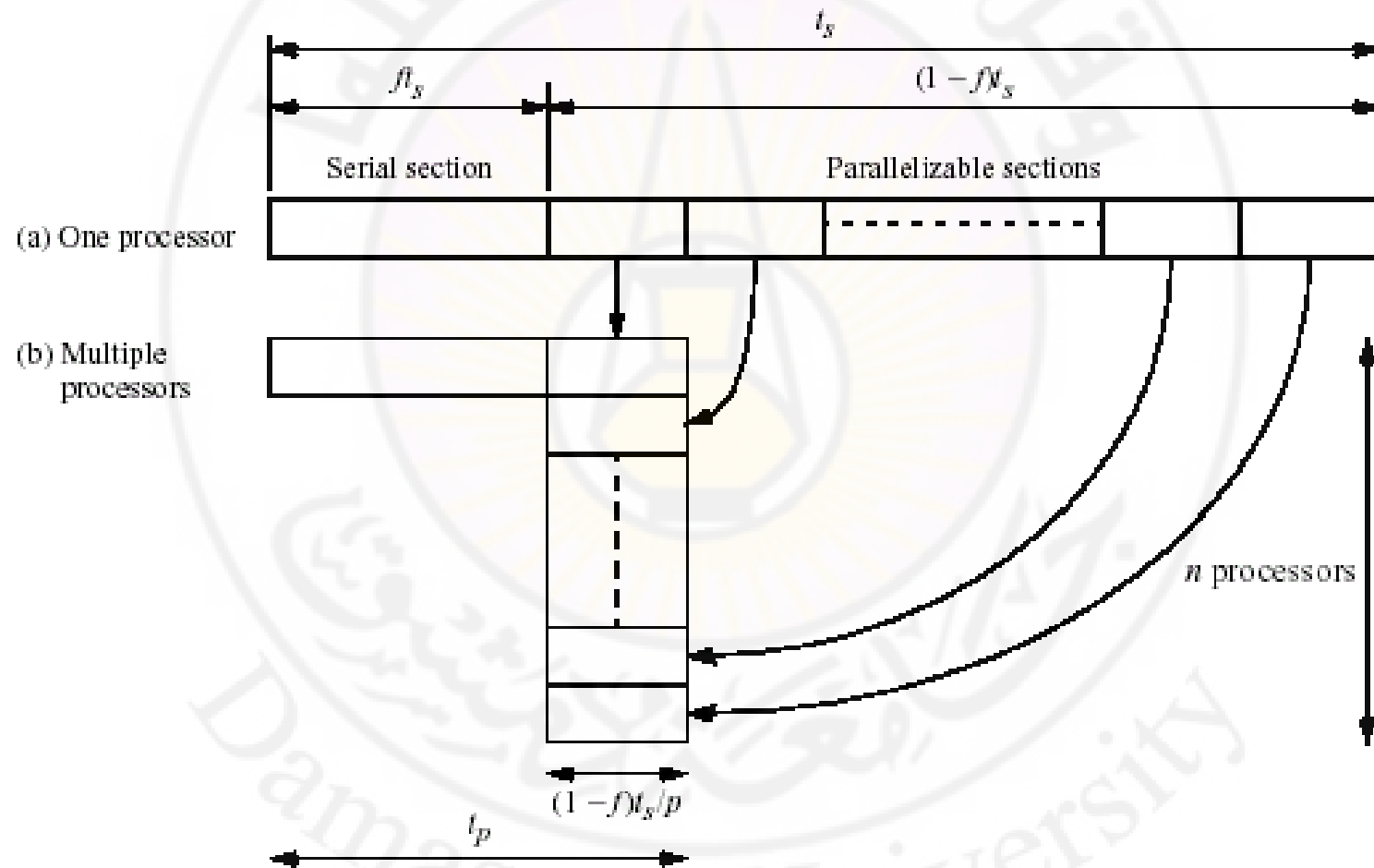
كلفة البرنامج التفرعي $t_p \times p$

$$Cost = (t_s \times p) / S(p) = t_s / E$$

تأخذ الكلفة قيمة أمثلية عندما تكون قيمتها باستخدام حاسب متعدد المعالجات مساوية لقيمتها باستخدام حاسب وحيد المعالج.

الاستجابة العظمى – قانون Amdahl

Maximum Speedup - Amdahl's law



الاستجابة العظمى – قانون Amdahl

Maximum Speedup - Amdahl's law

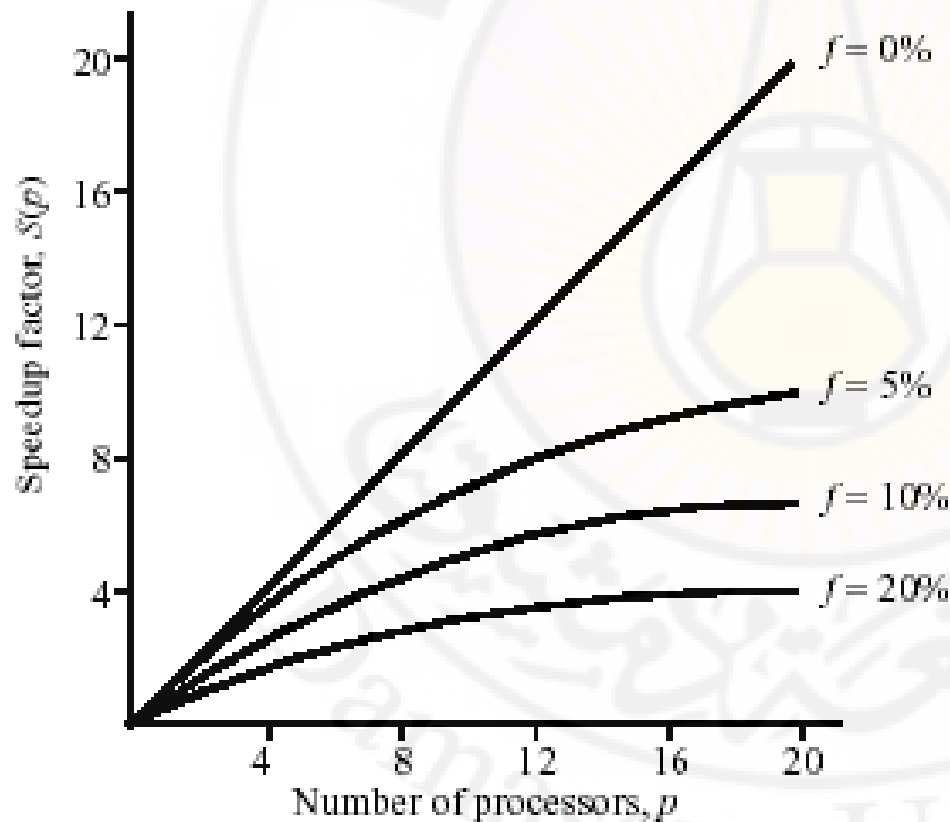
ليصبح معامل الاستجابة مساوياً لـ:

$$S(p) = \frac{t_s}{ft_s + (1 - f)t_s/p} = \frac{p}{1 + (p - 1)f}$$

تسمى هذه المعادلة بقانون Amdahl.

العلاقة بين الاستجابة وعدد المعالجات

Speedup against number of processors



يبقى معامل الاستجابة محدوداً بالقيمة $1/f$ حتى لو توفر لدينا عدد غير متناهي من المعالجات.

مثال:

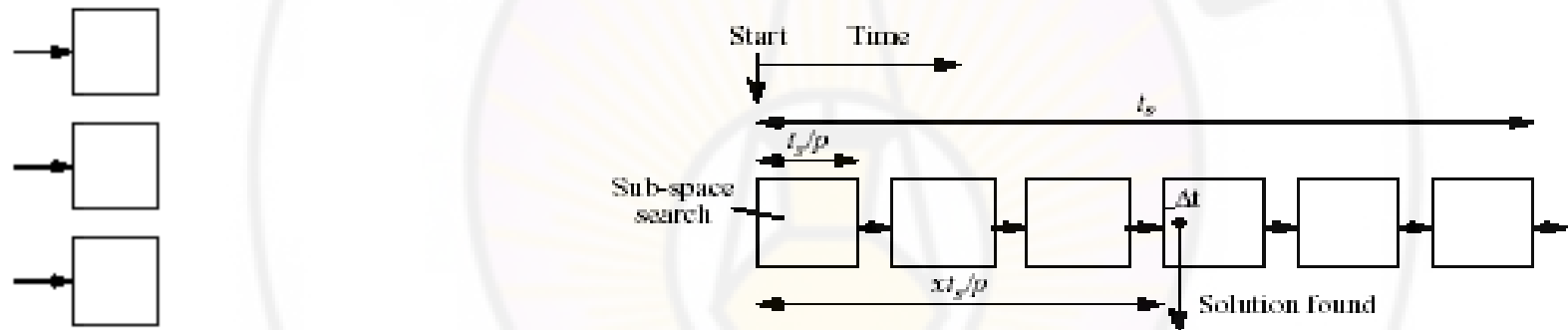
إذا كان الجزء التسلسلي من برنامج تفرعي يشكل 5%، فإن قيمة الاستجابة العظمى لن تتعدى القيمة 20 بغض النظر عن عدد المعالجات المستخدمة.

$$S(p) \underset{p \rightarrow \infty}{=} \frac{1}{f}$$

الاستجابة اللاخطية

Superlinear Speedup

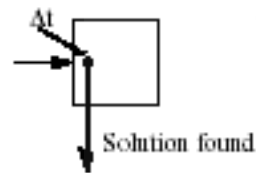
يمكن أن نحصل على معامل استجابة لاخطي عند تعاملنا مع خوارزميات البحث.



(a) Searching each sub-space sequentially

$$S(p) = \frac{\left(x \times \frac{t_s}{p}\right) + \Delta t}{\Delta t}$$

أسوأ حالة: الحل موجود في آخر مرحلة.



(b) Searching each sub-space in parallel

$$S(p) = \frac{\left(\frac{p-1}{p}\right) \times t_s + \Delta t}{\Delta t} \rightarrow \infty \text{ as } \Delta t \text{ tends to zero}$$

قابلية التوسع Scalability

وهي مفهوم مرتبط بالبنية الصلبة عموماً.

أما بالنسبة للخوارزميات التفرعية فهي تعني القدرة على التعامل مع عدد / حجم أكبر من المعطيات بعد التسبب بزيادة بسيطة أو ضعيفة في عدد خطوات الحساب.

يرتبط المفهوم السابق ارتباطاً وثيقاً بطبيعة المسألة المطلوب معالجتها.

ملاحظة هامة:

تؤدي زيادة حجم المسألة إلى حدوث تغيرات حسابية مختلفة.
مثال: يؤدي زيادة حجم المصفوفة مرتين في حالة جمع المصفوفات إلى زيادة كمية الحساب مرتين. في حين نفس الزيادة إلى زيادة كمية الحساب أربع مرات في حال ضرب المصفوفات.

قانون غوستافسون Gustafson's Law

يعتمد هذا القانون على مفهوم قابلية التوسع
زيادة حجم المسألة بشكل مضطرد مع زيادة أعداد المعالجات المستخدمة لحلها.
لا يوجد اي تزايد لحجم الجزء التسلسلي عند زيادة حجم المسألة.

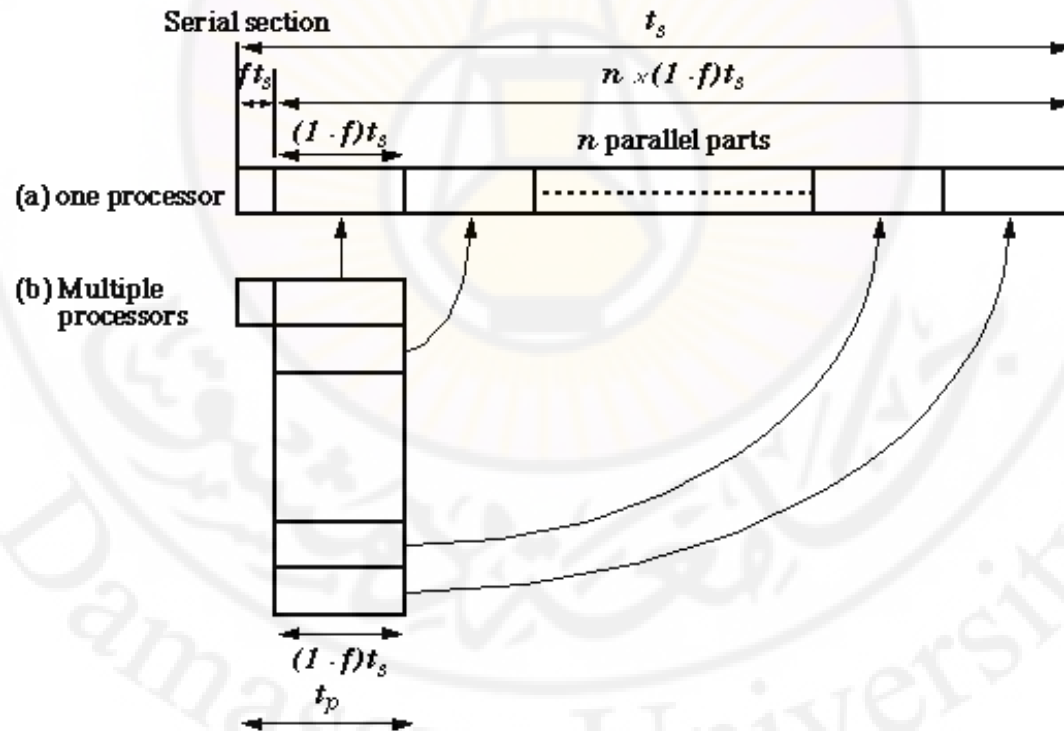


Figure 1.21 Gustafson's law

قانون غوستافسون

Gustafson's Law

- زيادة حجم المسألة بشكل مضطرد مع زيادة أعداد المعالجات المستخدمة لحلها.
- لا يوجد اي تزايد لحجم الجزء التسلسلي عند زيادة حجم المسألة.

$$S(n) = \frac{t_s}{t_p} = \frac{ft_s + n(1 - f)t_s}{ft_s + (1 - f)t_s} = n + f(1 - n)$$

البرمجة باستخدام تبادل الرسائل Message-Passing Computing

- أساسيات البرمجة باستخدام تبادل الرسائل - Basics of Message-Passing Programming
- استخدام تجمعات محطات العمل - Using Workstation Clusters
- تقييم أداء البرامج التفرعية - Evaluating Parallel Programs

الخيارات المتاحة

Programming Options

يمكن تصميم نظام متعدد المعالجات معتمد على تبادل الرسائل باستخدام أحد الوسائل التالية:

□ تصميم لغة برمجة تفرعية. مثال: OCCAM + Transputers.

□ توسيع قواعد التعامل مع لغة برمجة تسلسلية موجودة من قبل بحيث تصبح قادرة على تبادل الرسائل. مثال: اللغتين: CC+, Fortran M.

□ تزويد إحدى اللغات الموجودة حالياً بمكتبة من التوابع اللازمة لتبادل الرسائل. وهو الخيار الذي سنقوم بالتعامل معه من خلال دروسنا.

أساسيات البرمجة باستخدام تبادل الرسائل باستخدام مكتبة على مستوى المستثمر

Basics of Message-Passing Programming using user-level libraries

نحتاج إلى وسيلتين أساسيتين، هما:

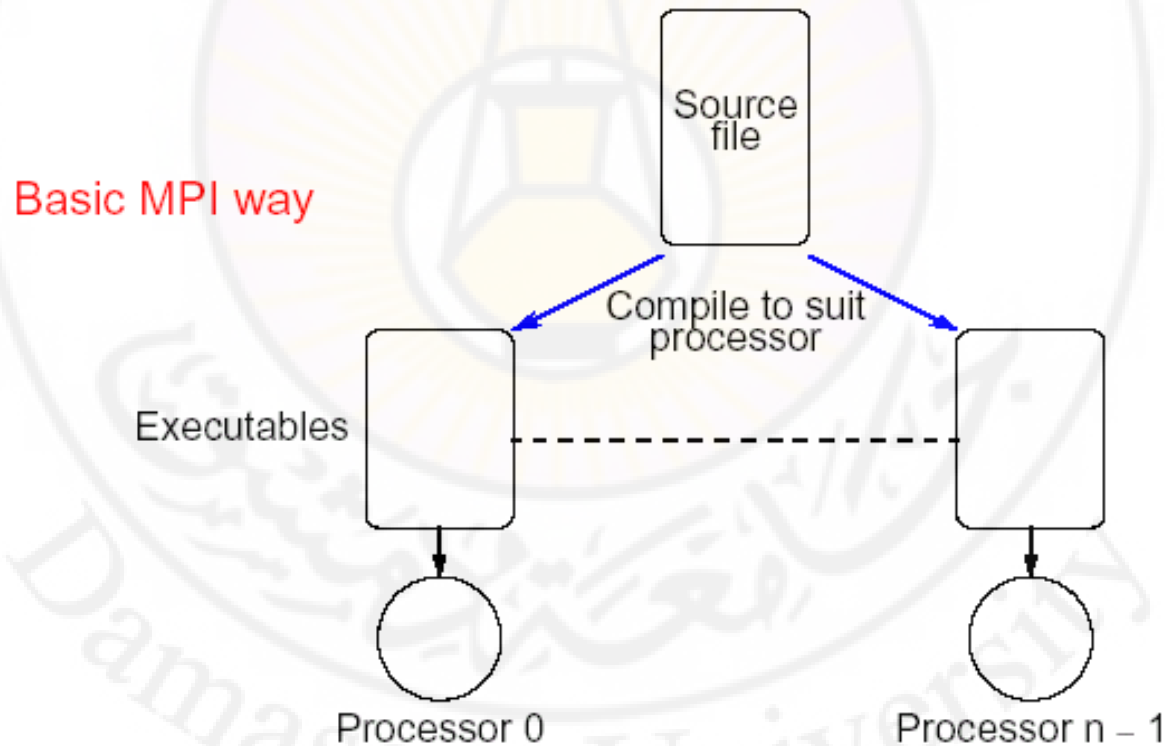
□ طريقة لإنشاء إجراءات منفصلة ليجري تنفيذها على حواسيب مختلفة.

□ طريقة لإرسال واستقبال الرسائل.

نموذج البرمجة SMPD

Single Program Multiple Data (SMPD) Model

تدمج عدة إجراءات لتشكل برنامجاً واحداً. ويجري ضمن البرنامج الاستعانة بتعليمات تحكم من أجل اختيار الأجزاء المناسبة لكل معالج لتنفيذه. تبدأ كافة البرامج التنفيذية معاً. تسمى هذه الطريقة طريقة الانشاء الثابت للإجراءات Static process creation.

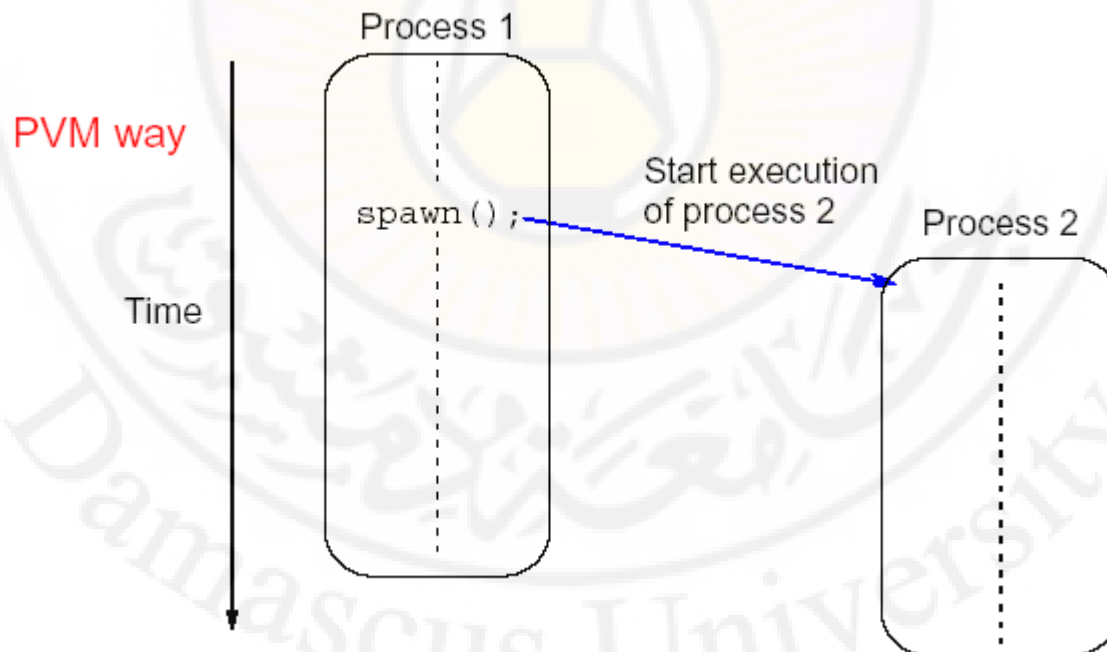


نموذج البرمجة MPMD

Multiple Program Multiple Data Model

نخصص برامج منفصلة لكل معالج. وغالباً ما نستخدم نموذج البرمجة الزبون / المخدم Server / Client حيث ينفذ أحد المعالجات الاجراء المخدم وتنفيذ البقية الاجراءات الزبونة.

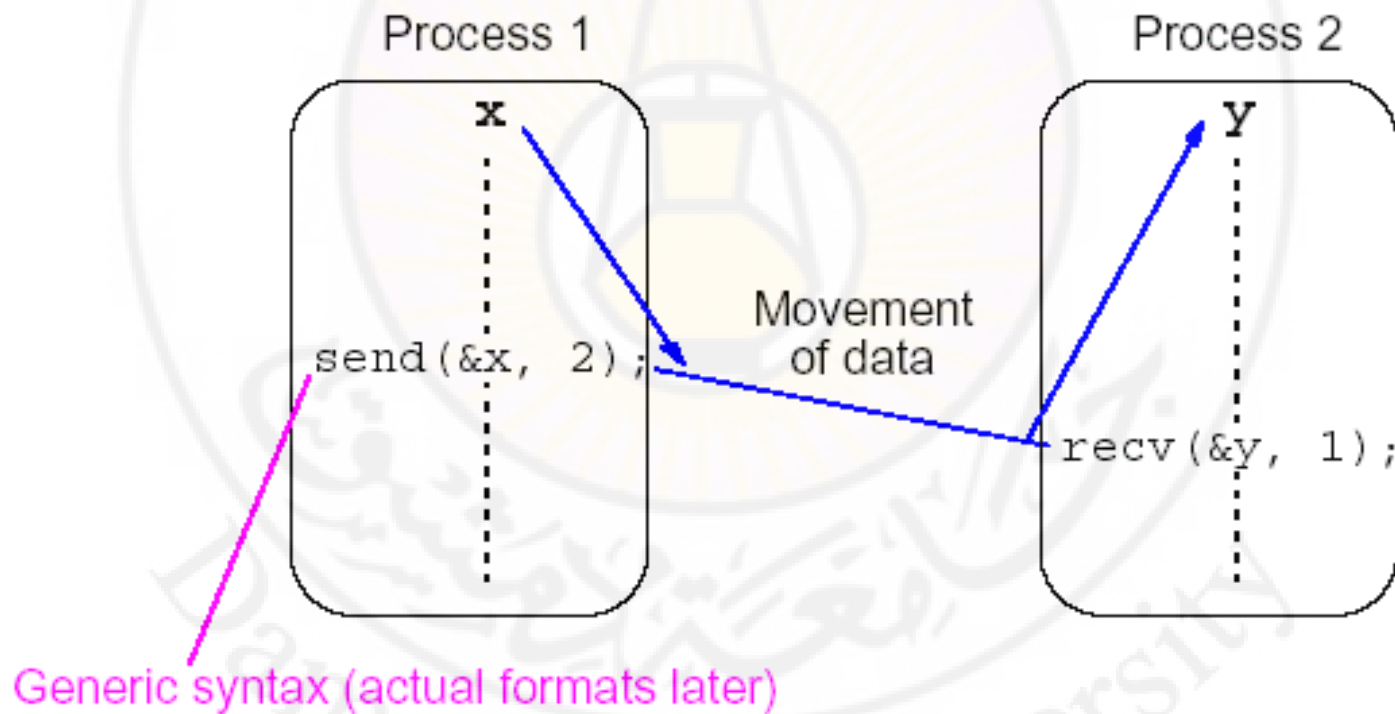
نسمي هذا الاسلوب طريقة الانشاء الديناميكي للاجراءات Dynamic process creation.



توابع الارسال والاستقبال الأساسية

Basic Send and Receive Routines

يجري تمرير الرسائل بين الاجراءات باستخدام التابعين `send(parameters)` و `recv(parameters)`.



التبادل المتزامن للرسائل

Synchronous Message Passing

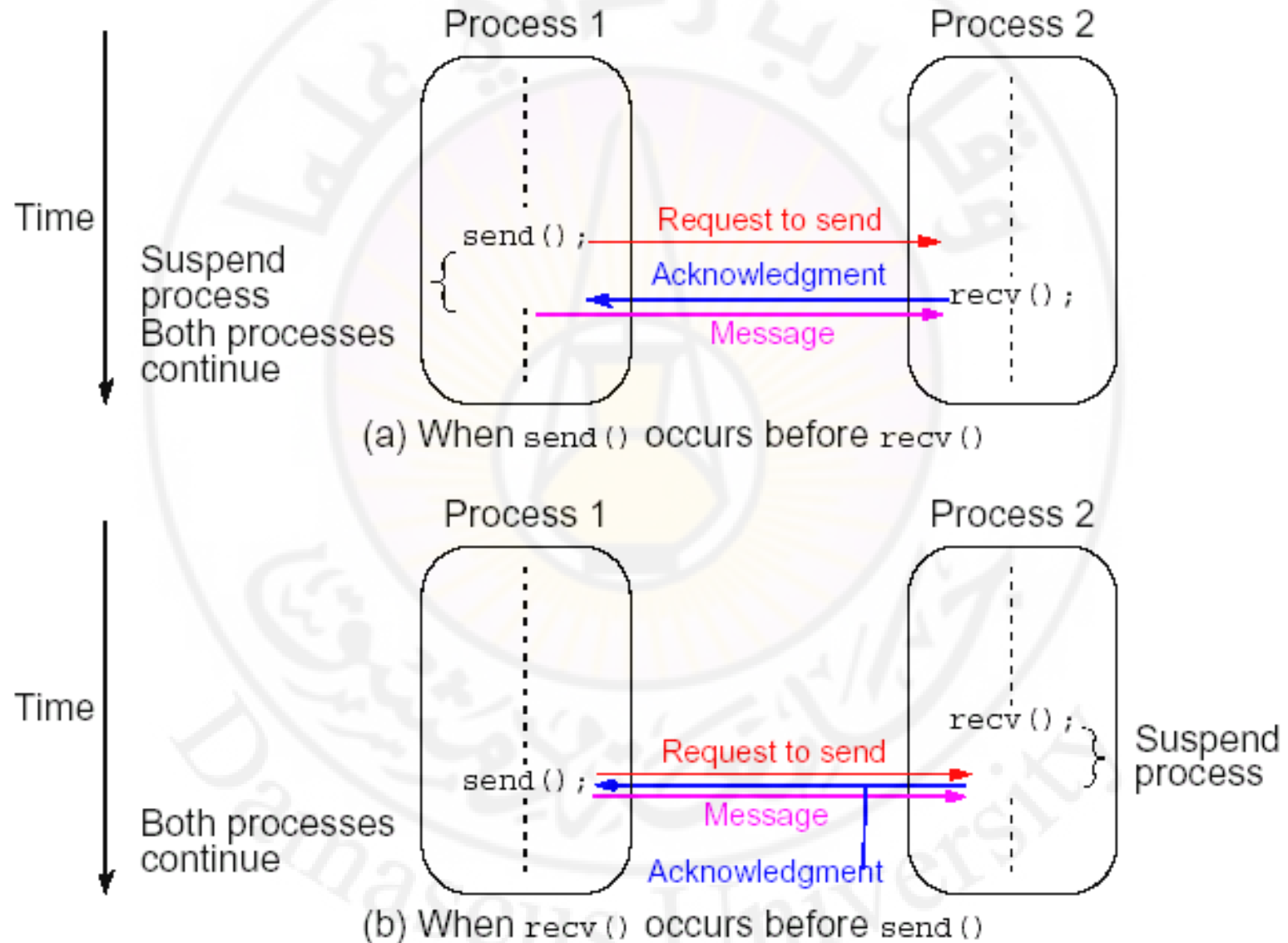
يُجري ذلك بواسطة نوع من التوابع التي تعيد التحكم إلى البرامج بعد انتهاء تبادل الرسائل تماماً.

□ يقوم تابع الإرسال المتزامن بالانتظار حتى يتأكد من إمكانية قبول الرسالة كاملة من قبل الإجراء المستقبل قبل إرسال الرسالة فعلياً.

□ يقوم تابع الاستقبال المتزامن بالانتظار حتى تصل الرسالة المتوقعة.

تفيد توابع تبادل الرسائل المتزامنة بالقيام في عمليتين، هما: نقل المعطيات Transfer data والمزامنة بين الإجراءات Synchronization.

الاستدعاءات `recv()` و `send()`



التبادل غير المتزامن للرسائل

Asynchronous Message Passing

يجري ذلك باستخدام توابع لا تقوم بانتظار أي شيء لاتمام عملها قبل إعادة التحكم إلى البرنامج.

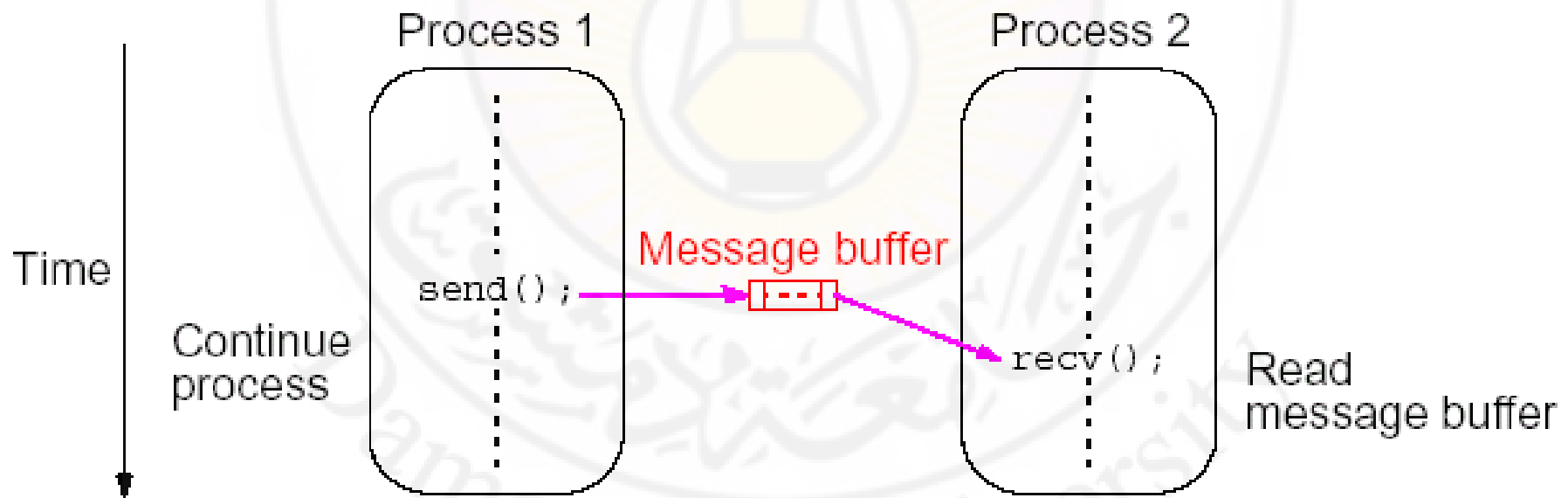
تتطلب هذه التوابع توفر إمكانية للتخزين المحلي للرسائل.

لا تسمح هذه التوابع بالزمالة بين الاجراءات، لكنها تسمح لها بالتحرك قدماً بأسرع ما يمكن. لكن يجب التعامل معها بحذر.

كيف تعود توابع تبادل الرسائل قبل انتهاء التبادل

How message-passing routines can return before message transfer completed

نحتاج إلى مكان تخزين مؤقت Message buffer بين المصدر والهدف للاحتفاظ بالرسائل.



تحويل التوابيع غير المتزامنة إلى توابيع متزامنة

Asynchronous routines changing to synchronous routines

عندما تكتمل الأحداث المحلية ويصبح بإمكان الرسالة متابعة طريقها بأمان، يمكن للإجراء المرسل أن يتابع عمله.

غالباً ما تكون أماكن التخزين المؤقت Buffers ذات سعة محدودة، مما يؤدي إلى الوصول إلى نقطة يتوقف عندها تابع الإرسال بسبب ذلك.

عند توقف تابع الإرسال حتى يتوفر مكان ضمن مكان التخزين المؤقت، يعمل تابع الإرسال كما لو أنه تابع متزامن.

دليل الرسالة Message Tag

ويستخدم للتمييز بين الأنواع المختلفة للرسائل المرسل.

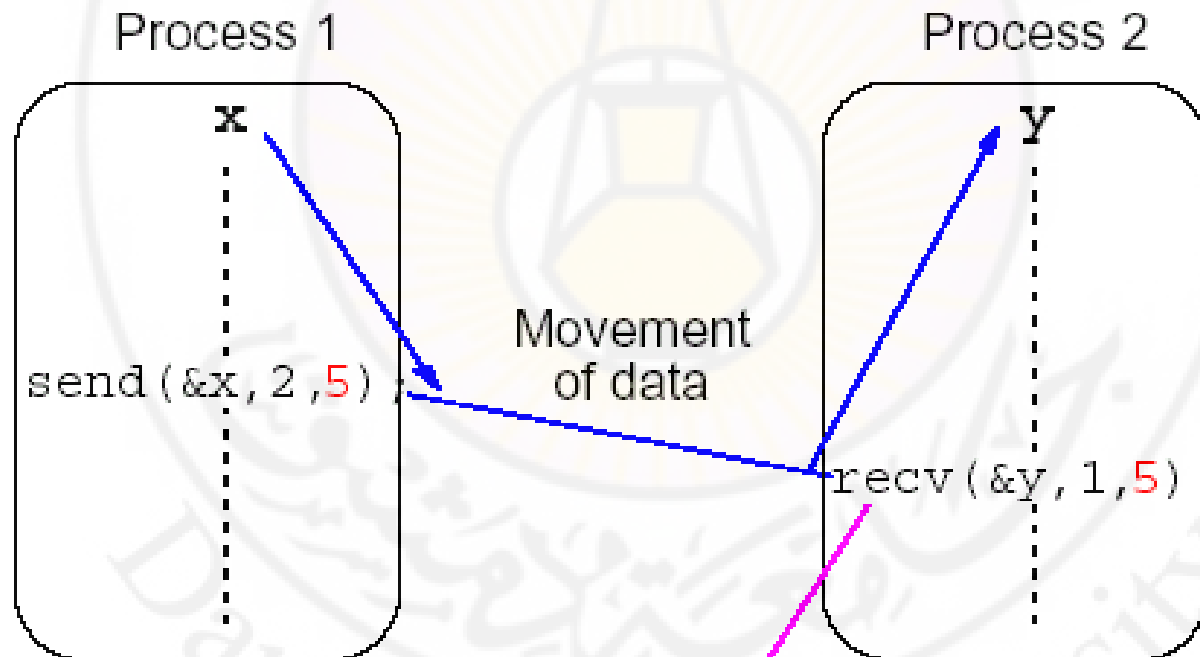
يجري تحميل دليل الرسالة Message tag وارساله مع الرسالة.

إذا لم يكن مطلوباً التعامل مع نوع معين من الرسائل، يمكن عندها استخدام الدليل العام Wild card بحيث يستطيع تابع الاستقبال `recv()` التعامل مع أي رسالة مرسل بواسطة التابع `send()`.

دليل الرسالة (مثال)

Message Tag Example

يمكن ارسال الرسالة x ذات الدليل ٥ من الاجراء المصدر رقم (١) إلى الاجراء الهدف رقم (٢) وتخزينها في الموضع y على الشكل التالي:



Waits for a message from process 1 with a tag of 5

توابع تمرير الرسائل إلى مجموعة

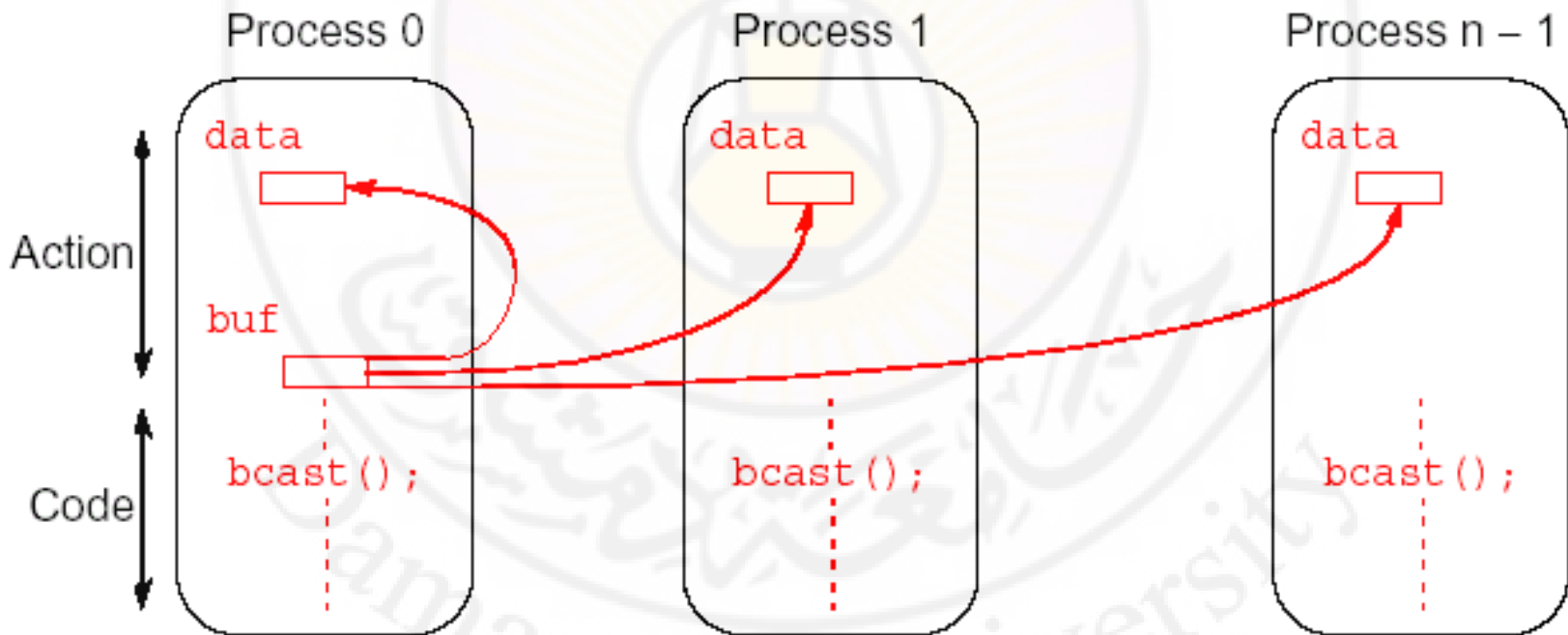
Group message passing routines

إضافة إلى توابع تبادل الرسائل من نقطة إلى أخرى، يتوفر لدينا عدد آخر من التوابع لإرسال رسائل إلى مجموعة من الإجراءات أو استقبال رسائل منها.

تتميز هذه التوابع بفعالية كبيرة مقارنة بتوابع تبادل الرسائل من نقطة إلى أخرى لكنها ليست ضرورية.

الإذاعة Broadcast

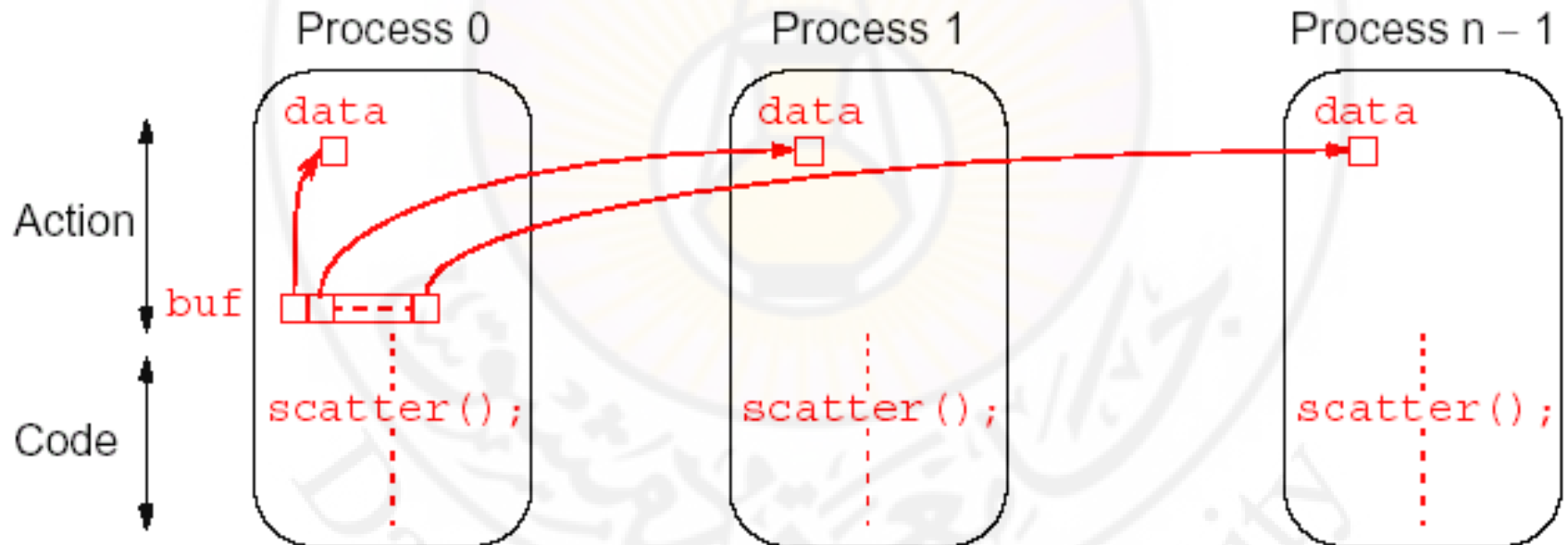
إرسال نفس الرسالة إلى جميع الإجراءات التي لها علاقة بالمسألة المطلوب معالجتها.
Multicast – ويعني إرسال نفس الرسالة إلى مجموعة محددة ومعروفة من الإجراءات.



MPI form

التوزيع Scatter

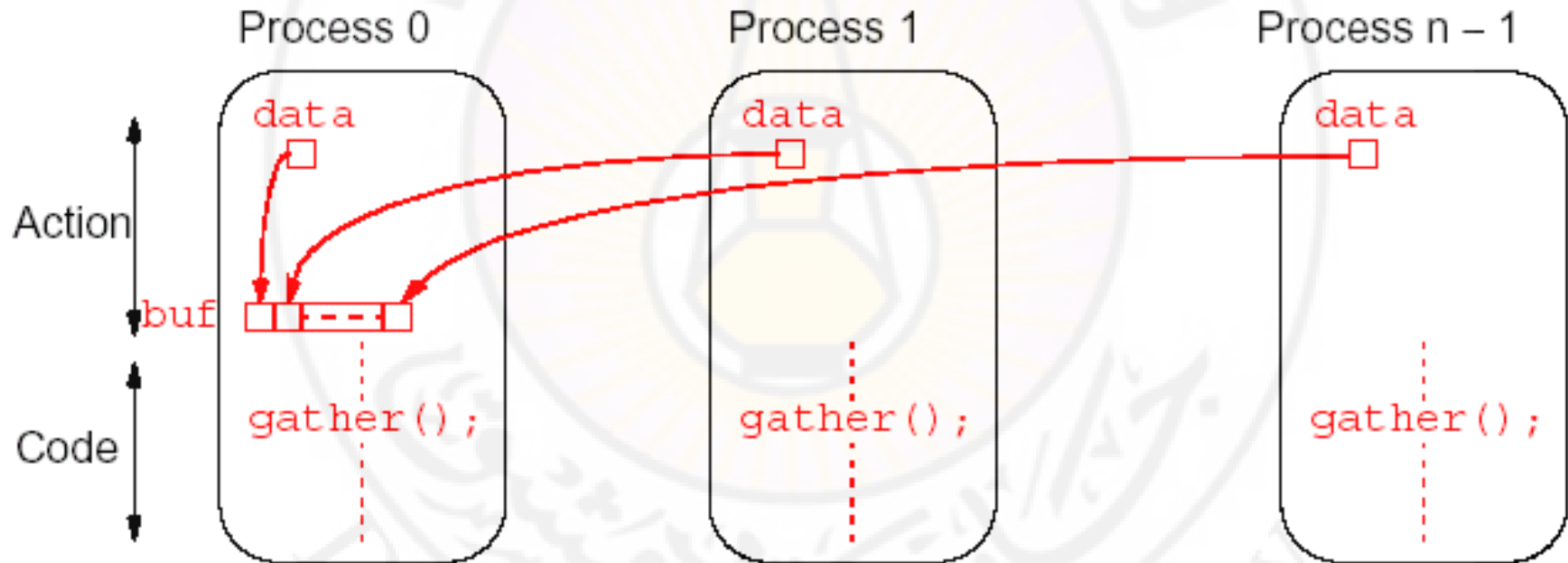
ارسال كل عنصر من عناصر مصفوفة موجودة ضمن الاجراء الرئيسي إلى إجراء منفصل. أي، نرسل العنصر رقم i إلى الإجراء رقم i .



MPI form

التجميع Gather

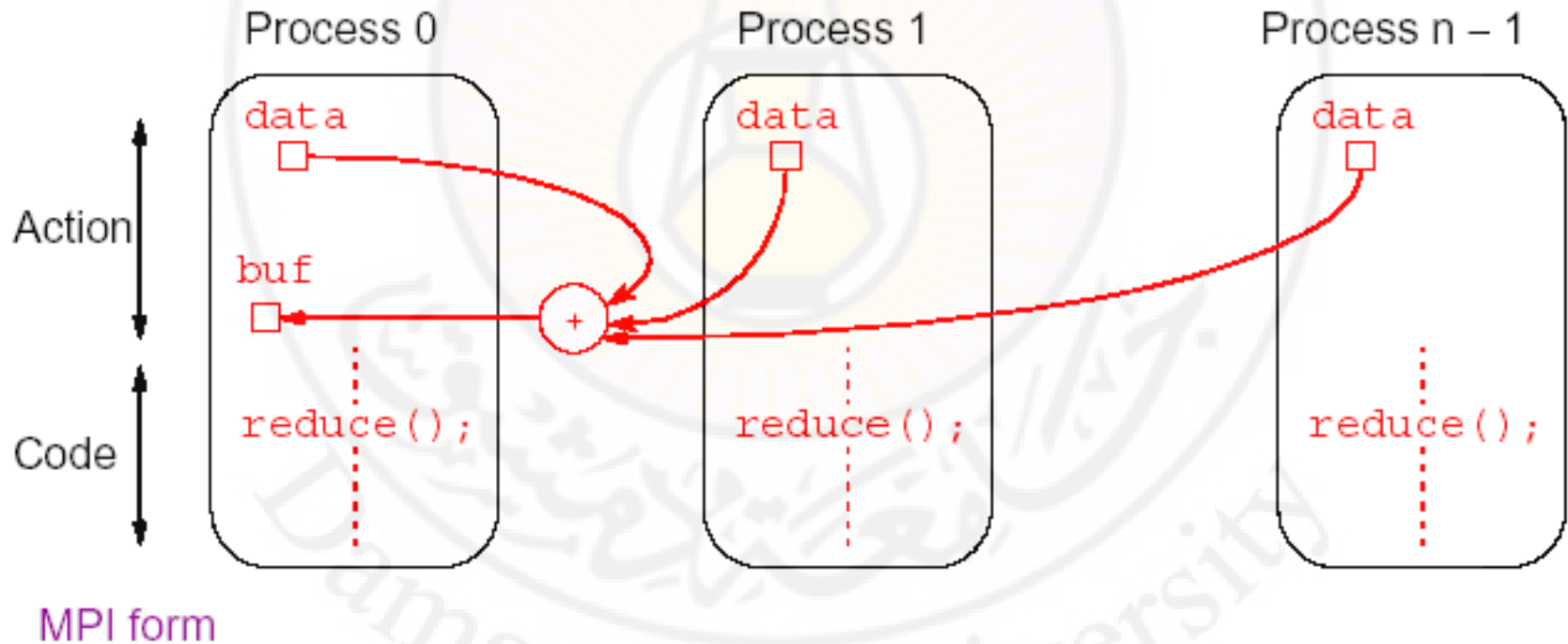
يقوم إجراء وحيد بجمع القيم الإفرادية الموزعة ضمن مجموعة من الإجراءات.



MPI form

التقليل Reduce

تشبه عملية التجميع ولكن مع إضافة عملية حسابية أو منطقية إليها ضمن الاجراء الرئيسي. كما هو موضح في المثال التالي.



الآلة التفرعية الافتراضية

Parallel Virtual Machine

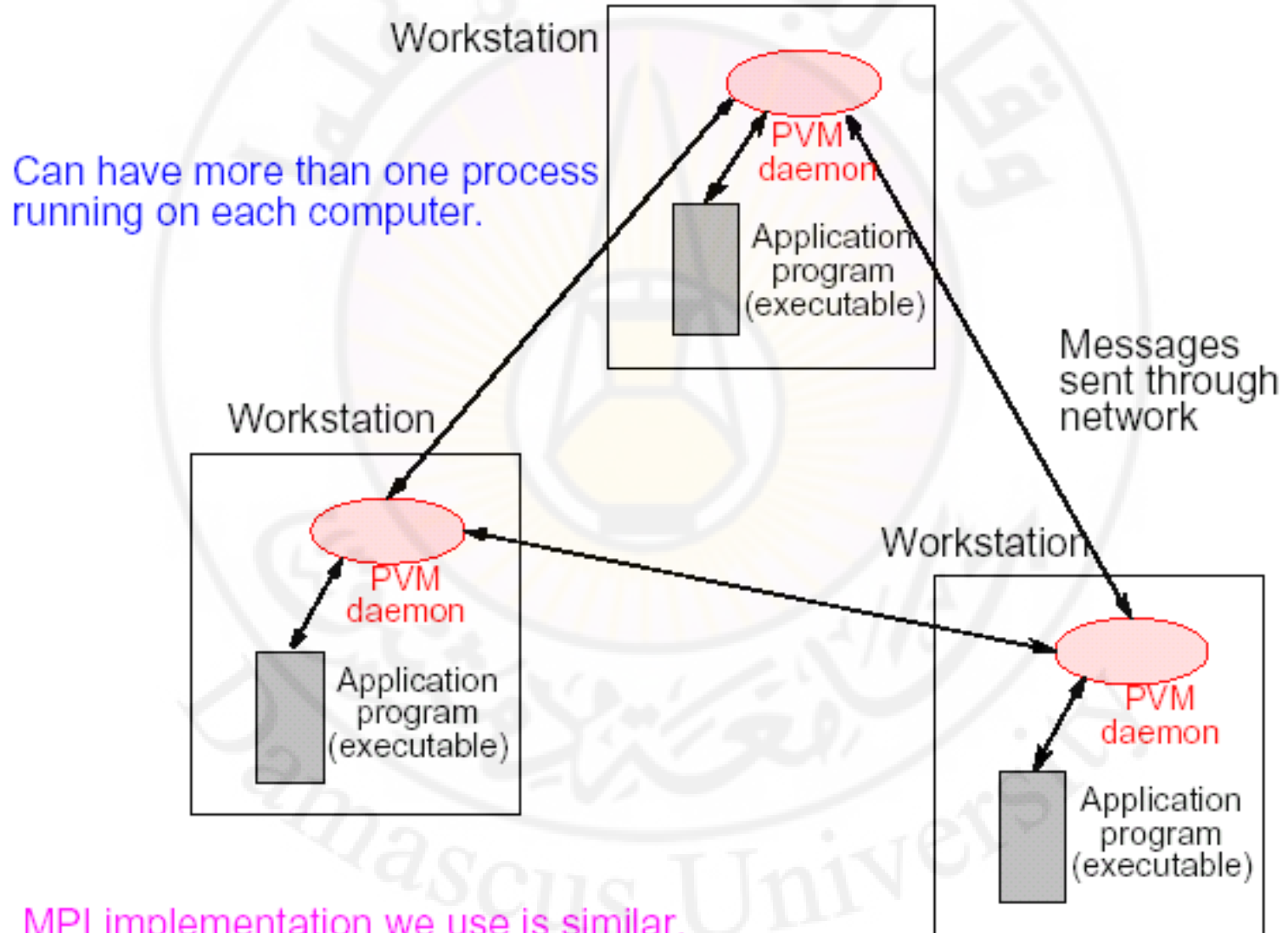
وهي المحاولة الأولى التي جرى تبنيها من أجل استخدام محطات العمل كما لو أنها بيئة عمل متعددة المعالجات. جرى تطوير هذه الآلة الافتراضية من قبل مختبرات Oak Ridge National Laboratories وهي متاحة مجاناً.

يقوم المبرمج بتقسيم المسألة المطلوب حلها إلى عدة برامج (برنامج رئيسي Master ومجموعة من البرامج الفرعية Slaves).

تجري ترجمة كل برنامج لتنفيذه على نوع محدد من الحواسيب.

يجب تعريف الحواسيب المطلوب استخدامها لحل مسألة ما مقدماً قبل تنفيذ البرامج (ضمن الملف hostfile).

يجري تمرير الرسائل بين الحواسيب بواسطة الاجراءات PVM daemon المثبتة
بواسطة PVM على الحواسيب المكونة للآلة الافتراضية.



MPI implementation we use is similar.

توابع PVM لتبادل الرسائل

PVM Message-Passing Routines

كافة توابع الارسال في PVM هي توابع غير موقفة nonblocking (أو غير متزامنة).

يمكن لتوابع الاستقبال في PVM أن تكون موقفة blocking (متزامنة) أو غير موقفة nonblocking.

يمكن استخدام دليل الرسالة Message tag و الحروف العامة Wild cards مع هذه التوابع.

The background of the slide features a large, faint watermark of the Damascus University logo. The logo is circular, with the university's name in Arabic 'جامعة دمشق' at the top and 'Damascus University' at the bottom. In the center is a stylized sunburst or star symbol.

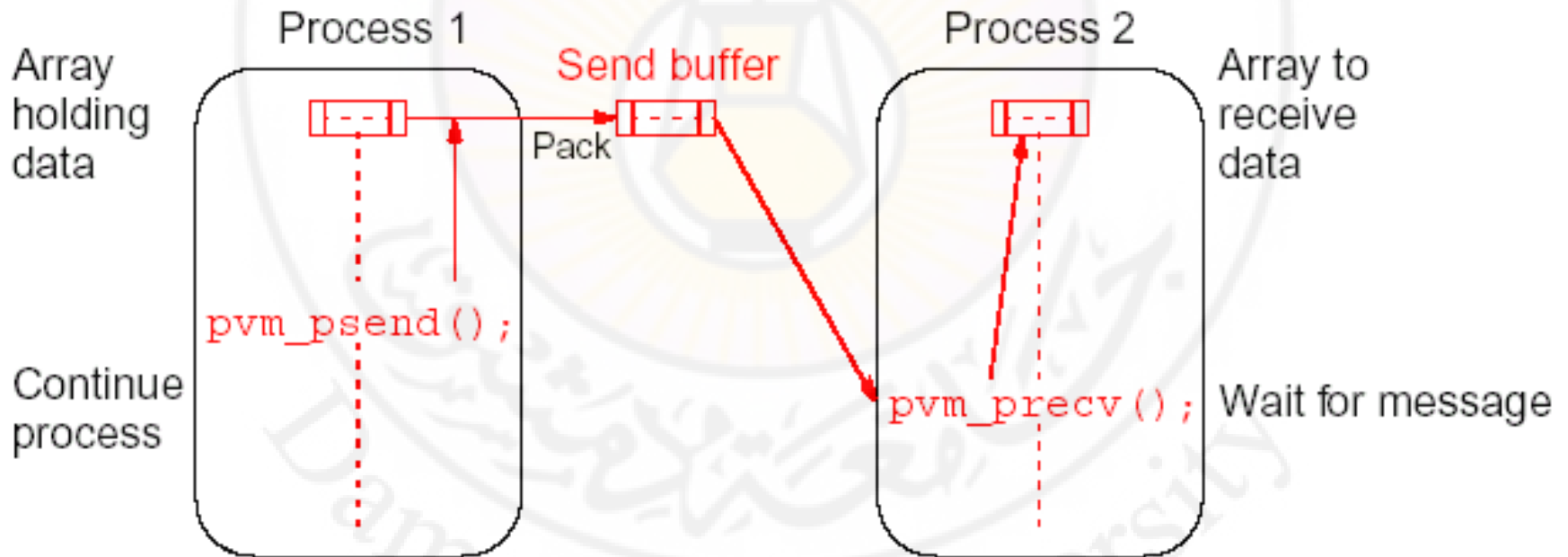
توابع PVM الاساسية لتبادل الرسائل

Basic PVM Message-Passing Routines

استدعائي النظام `pvm_psend()` و `pvm_precv()`

`pvm_psend()` and `pvm_precv()` system calls

يمكن استخدامها إذا كانت المعطيات المطلوب إرسالها هي عبارة عن قائمة من معطيات من نفس النمط.



قائمة وسطاء التابعين السابقين

Full list of parameters for the two previous system calls

Pvm_psend(int dest_tid, int msgtag, char *buf, int len, int datatype)

Pvm_precv(int source_tid, int msgtag, char *buf, int len, int datatype)

ارسال معطيات مؤلفة من عدة أنماط

Sending Data Composed of Various Types

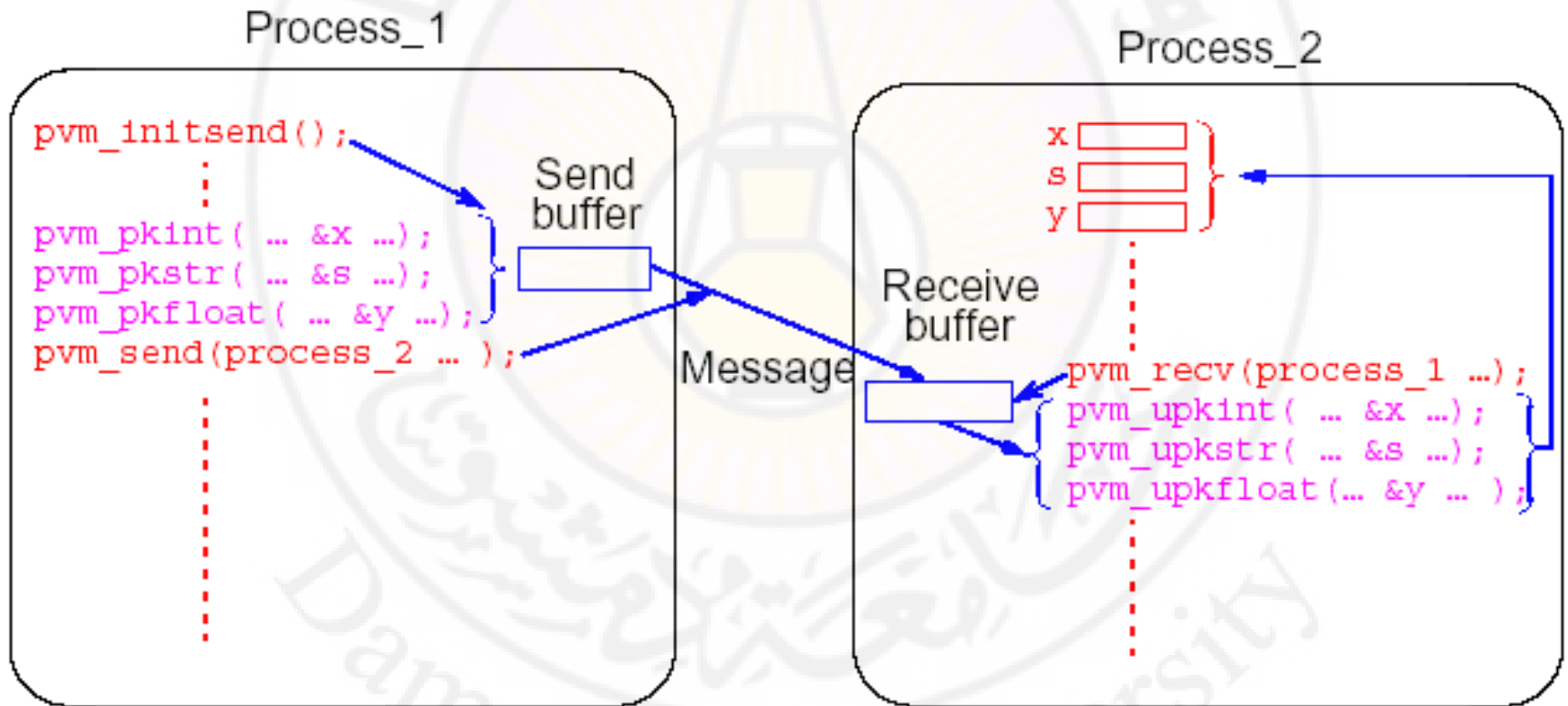
تحزم المعطيات ضمن خزان مؤقت buffer قبل ارسالها.

يقوم الاجراء المستقبل بفك حزمة المعطيات المستقبلية ضمن خزان الاستقبال وفقاً للصيغة التي اعتمدت أثناء الحزم.

يتوفر توابع للحزم وفك الحزم packing/unpacking routines من أجل كل نمط من أنماط المعطيات.

ارسال معطيات مؤلفة من عدة أنماط (مثال)

Sending Data Composed of Various Types (Example)



التوابع Broadcast, Multicast, Scatter, Gather, Reduce

pvm_bcast()
pvm_scatter()
pvm_gather()
pvm_reduce()

تعمل هذه التوابع مع مجموعة من الاجراءات.

يمكن لاجراء أن ينضم إلى مجموعة باستخدام التابع pvm_joingroup().

ليست العملية pvm_mcast() عملية على مجموعة من الاجراءات.

مثال على برنامج PVM

Master

```
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
#define SLAVE "spsum"
#define PROC 10
#define NELEM 1000
main() {
    int mytid,tids[PROC];
    int n = NELEM, nproc = PROC;
    int no, i, who, msgtype;
    int data[NELEM],result[PROC],tot=0;
    char fn[255];
    FILE *fp;
    mytid=pvm_mytid();/*Enroll in PVM */

    /* Start Slave Tasks */
    no=
    pvm_spawn(SLAVE,(char**)0,0,"",nproc,tids);
    if (no < nproc) {
        printf("Trouble spawning slaves \n");
        for (i=0; i<no; i++) pvm_kill(tids[i]);
        pvm_exit(); exit(1);
    }

    /* Open Input File and Initialize Data */
    strcpy(fn,getenv("HOME"));
    strcat(fn,"/pvm3/src/rand_data.txt");
    if ((fp = fopen(fn,"r")) == NULL) {
        printf("Can't open input file %s\n",fn);
        exit(1);
    }
    for(i=0;i<n;i++) fscanf(fp,"%d",&data[i]);
    printf("%d from %d\n",result[who],who);
```

Slave

```
#include <stdio.h>
#include "pvm3.h"
#define PROC 10
#define NELEM 1000

main() {
    int mytid;
    int tids[PROC];
    int n, me, i, msgtype;
    int x, nproc, master;
    int data[NELEM], sum;
```

مثال على برنامج PVM (تتمة)

```

/* Open Input File and Initialize Data */
strcpy(fn, getenv("HOME"));
strcat(fn, "/pvm3/src/rand_data.txt");
if ((fp = fopen(fn, "r")) == NULL) {
    printf("Can't open input file %s\n", fn);
    exit(1);
}
for(i=0; i<n; i++) fscanf(fp, "%d", &data[i]);

/* Broadcast data To slaves*/
pvm_initsend(PvmDataDefault);
msgtype = 0;
pvm_pkint(&nproc, 1, 1);
pvm_pkint(tids, nproc, 1);
pvm_pkint(&n, 1, 1);
pvm_pkint(data, n, 1);
pvm_mcast(tids, nproc, msgtag);

/* Get results from Slaves*/
msgtype = 5;
for (i=0; i<nproc; i++){
    pvm_recv(-1, msgtype);
    pvm_upkint(&who, 1, 1);
    pvm_upkint(&result[who], 1, 1);
    printf("%d from %d\n", result[who], who);
}

/* Compute global sum */
for (i=0; i<nproc; i++) tot += result[i];
printf ("The total is %d.\n\n", tot);

pvm_exit(); /* Program finished. Exit PVM */
return(0);
}

mytid = pvm_mytid();

/* Receive data from master */
msgtype = 0;
pvm_recv(-1, msgtype);
pvm_upkint(&nproc, 1, 1);
pvm_upkint(tids, nproc, 1);
pvm_upkint(&n, 1, 1);
pvm_upkint(data, n, 1);

/* Determine my tid */
for (i=0; i<nproc; i++)
    if(mytid==tids[i])
        {me = i; break;}

/* Add my portion Of data */
x = n/nproc;
low = me * x;
high = low + x;
for(i = low; i < high; i++)
    sum += data[i];

/* Send result to master */
pvm_initsend(PvmDataDefault);
pvm_pkint(&me, 1, 1);
pvm_pkint(&sum, 1, 1);
msgtype = 5;
master = pvm_parent();
pvm_send(master, msgtype);

/* Exit PVM */
pvm_exit();
return(0);
}

```

Broadcast data

Receive results

MPI **(Message Passing Interface)**

- ❑ Message passing library standard developed by group of academics and industrial partners to foster more widespread use and portability.
- ❑ Defines routines, not implementation.
- ❑ Several free implementations exist.

MPI

Process Creation and Execution

- ❑ Purposely not defined - Will depend upon implementation.
- ❑ Only static process creation supported in MPI version 1. All processes must be defined prior to execution and started together.
- ❑ Originally SPMD model of computation.
- ❑ MPMD also possible with static creation - each program to be started together specified.

Communicators

- Defines scope of a communication operation.
- Processes have ranks associated with communicator.
- Initially, all processes enrolled in a “universe” called `MPI_COMM_WORLD`, and each process is given a unique rank, a number from 0 to $p - 1$, with p processes.
- Other communicators can be established for groups of processes.

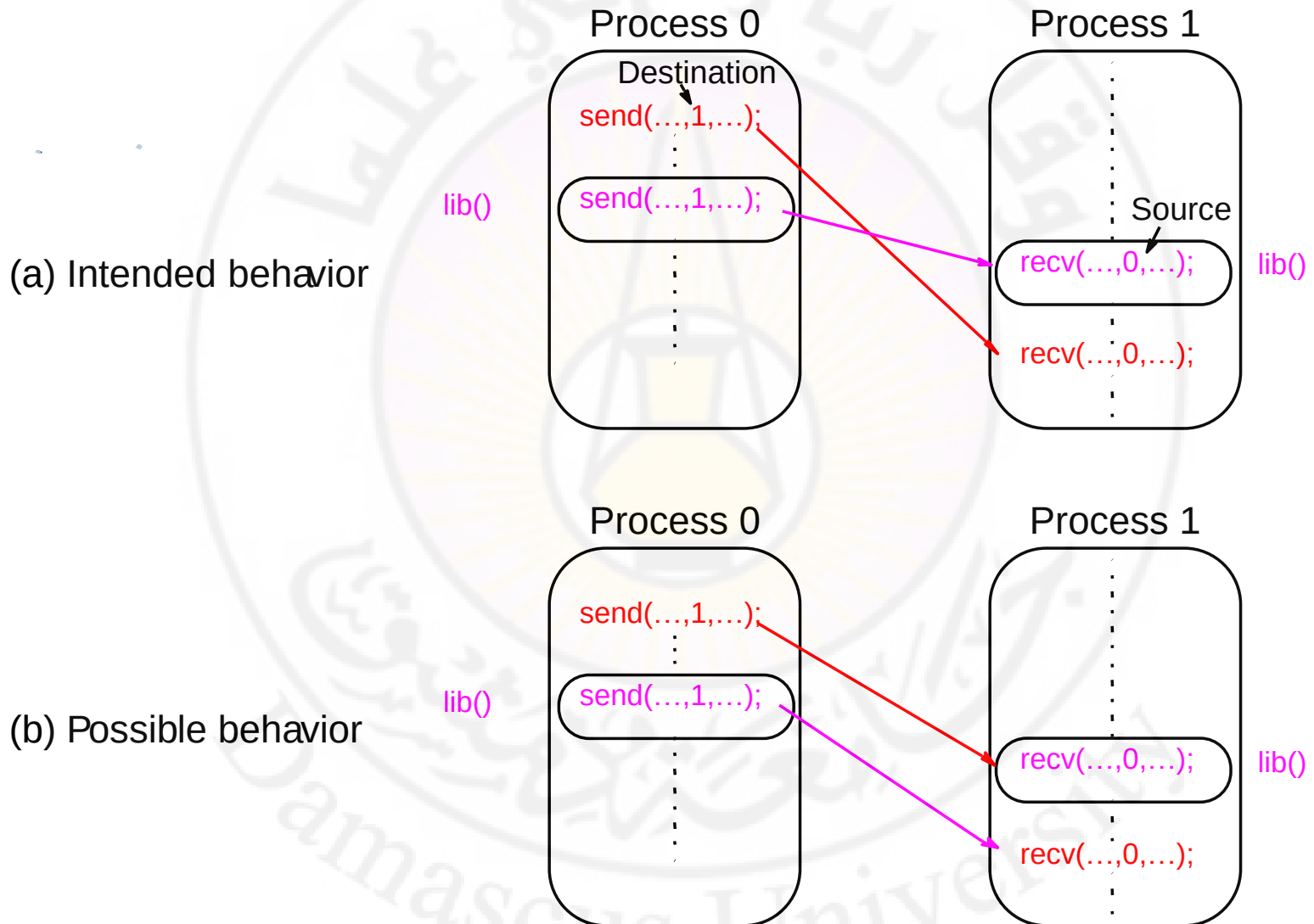
Using SPMD Computational Model

```
main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    .
    .
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */

    if (myrank == 0)
        master();
    else
        slave();
    .
    .
    MPI_Finalize();
}
```

where master() and slave() are to be executed by master process and slave process, respectively.

Unsafe message passing - Example



MPI Solution

“Communicators”

- ❑ Defines a communication domain - a set of processes that are allowed to communicate between themselves.
- ❑ Communication domains of libraries can be separated from that of a user program.
- ❑ Used in all point-to-point and collective MPI message-passing communications.

Default Communicator `MPI_COMM_WORLD`

- ❑ Exists as first communicator for all processes existing in the application.
- ❑ A set of MPI routines exists for forming communicators.
- ❑ Processes have a “rank” in a communicator.

MPI Point-to-Point Communication

- ❑ Uses send and receive routines with message tags (and communicator).
- ❑ Wild card message tags available

MPI Blocking Routines

- Return when “locally complete” - when location used to hold message can be used again or altered without affecting message being sent.
- Blocking send will send message and return - does not mean that message has been received, just that process free to move on without adversely affecting message.

Parameters of blocking send

MPI_Send(buf, count, datatype, dest, tag, comm)

Address of
send buffer

Number of items
to send

Datatype of
each item

Rank of destination
process

Message tag

Communicator

Parameters of blocking receive

MPI_Recv(buf, count, datatype, src, tag, comm, status)

Address of
receive buffer

Maximum number
of items to receive

Datatype of
each item

Rank of source
process

Message tag

Communicator

Status
after operation

Example

To send an integer x from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */

if (myrank == 0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag,
             MPI_COMM_WORLD);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT,
             0, msgtag, MPI_COMM_WORLD, status);
}
```

MPI Nonblocking Routines

- ❑ **Nonblocking send** - `MPI_Isend()` - will return “immediately” even before source location is safe to be altered.
- ❑ **Nonblocking receive** - `MPI_Irecv()` - will return even if no message to accept.

Nonblocking Routine Formats

`MPI_Isend(buf, count, datatype, dest, tag, comm, request)`

`MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

Completion detected by `MPI_Wait()` and `MPI_Test()`.

`MPI_Wait()` waits until operation completed and returns then.

`MPI_Test()` returns with flag set indicating whether operation completed at that time.

Need to know whether particular operation completed.

Determined by accessing `request` parameter.

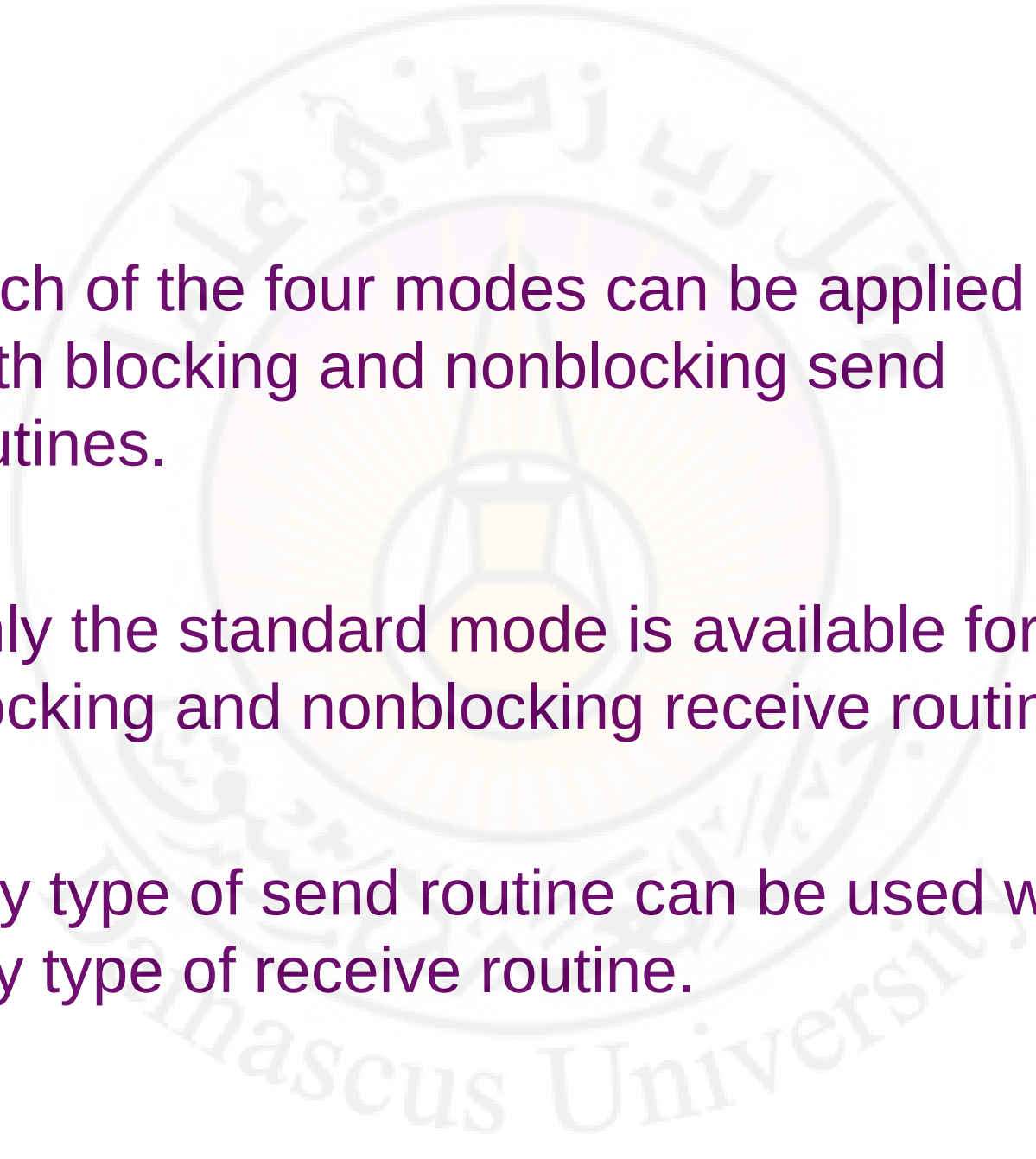
Example

To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```

Send Communication Modes

- ❑ **Standard Mode Send** - Not assumed that corresponding receive routine has started. Amount of buffering not defined by MPI. If buffering provided, send could complete before receive reached.
- ❑ **Buffered Mode** - Send may start and return before a matching receive. Necessary to specify buffer space via routine `MPI_Buffer_attach()`.
- ❑ **Synchronous Mode** - Send and receive can start before each other but can only complete together.
- ❑ **Ready Mode** - Send can only start if matching receive already reached, otherwise error. Use with care.

- 
- ❑ Each of the four modes can be applied to both blocking and nonblocking send routines.
 - ❑ Only the standard mode is available for the blocking and nonblocking receive routines.
 - ❑ Any type of send routine can be used with any type of receive routine.

Collective Communication

Involves set of processes, defined by an intra-communicator. Message tags not present. Principal collective operations:

- ❑ `MPI_Bcast()` - Broadcast from root to all other processes
- ❑ `MPI_Gather()` - Gather values for group of processes
- ❑ `MPI_Scatter()` - Scatters buffer in parts to group of processes
- ❑ `MPI_Alltoall()` - Sends data from all processes to all processes
- ❑ `MPI_Reduce()` - Combine values on all processes to single value
- ❑ `MPI_Reduce_scatter()` - Combine values and scatter results
- ❑ `MPI_Scan()` - Compute prefix reductions of data on processes

Example

To gather items from group of processes into process 0, using dynamically allocated memory in root process:

```
int data[10]; /*data to be gathered from
processes*/
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    MPI_Comm_size(MPI_COMM_WORLD, &grp_size); /*find
group size*/
    buf = (int *)malloc(grp_size*10*sizeof (int));
    /*allocate memory*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_
COMM_WORLD) ;
```

`MPI_Gather()` gathers from all processes, including root.

Barrier routine

- A means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.

Sample MPI program

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv)
{
    int myid, numprocs;
    int data[MAXSIZE], i, x, low, high, myresult, result;
    char fn[255];
    char *fp;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0) { /* Open input file and initialize data */
        strcpy(fn, getenv("HOME"));
        strcat(fn, "/MPI/rand_data.txt");
        if ((fp = fopen(fn, "r")) == NULL) {
            printf("Can't open the input file: %s\n\n", fn);
            exit(1);
        }
        for(i = 0; i < MAXSIZE; i++) fscanf(fp, "%d", &data[i]);
    }
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */
    x = n/numprocs; /* Add my portion Of data */
    low = myid * x;
    high = low + x;
    for(i = low; i < high; i++)
        myresult += data[i];
    printf("I got %d from %d\n", myresult, myid); /* Compute global sum */
    MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) printf("The sum is %d.\n", result);
    MPI_Finalize();
}
```

تقييم البرامج التفرعية

Evaluation Parallel Programs

نحسب عادة زمن التنفيذ التفرعي t_p كما يلي:

$$t_p = t_{comp} + t_{comm}$$

حيث يمثل الزمن t_{comp} زمن التنفيذ لأطول إجراء من الإجراءات التفرعية المرتبطة بالبرنامج المدروس.

ويمثل الزمن t_{comm} زمن الاتصالات وتبادل الرسائل الذي يرتبط بطول الرسائل المتبادلة.

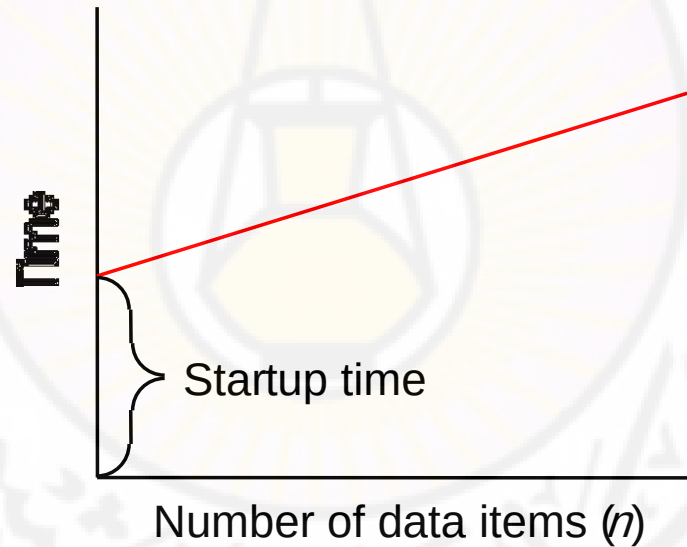
نفترض أنه يمكن عادة حساب زمن الاتصال t_{comm} تقريباً وفقاً للصيغة التالية:

$$t_{comm} = t_{startup} + n t_{data}$$

يمثل الزمن $t_{startup}$ زمن تحضير الرسالة Message latency الذي يشمل زمن حزم المعطيات عند الإرسال وفكها عند الاستقبال.

أما الزمن t_{data} فيمثل زمن نقل وحدة معطيات واحدة (زمن نقل word).

Idealized Communication Time



Benchmark Factors

With t_s , t_{comp} , and t_{comm} , can establish speedup factor and computation/communication ratio for a particular algorithm/implementation:

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{t_s}{t_{\text{comp}} + t_{\text{comm}}}$$

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}}$$

Both functions of number of processors, p , and number of data elements, n .

Factors give indication of scalability of parallel solution with increasing number of processors and problem size.

Computation/communication ratio will highlight effect of communication with increasing problem size and system size.

Broadcast/Gather

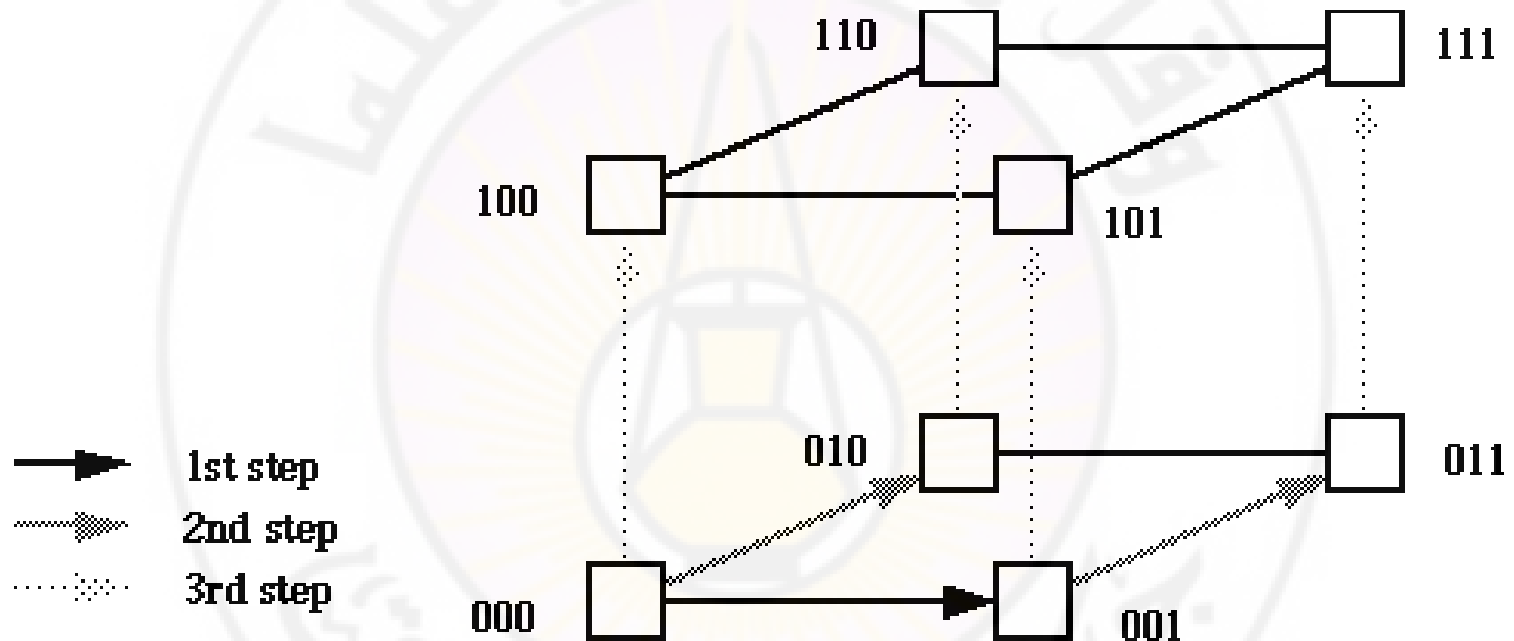


Figure 2.9 Broadcast in a three-dimensional hypercube

To broadcast to all nodes in an n -dimensional hypercube in $\log_2(n)$ steps

Broadcast/Gather

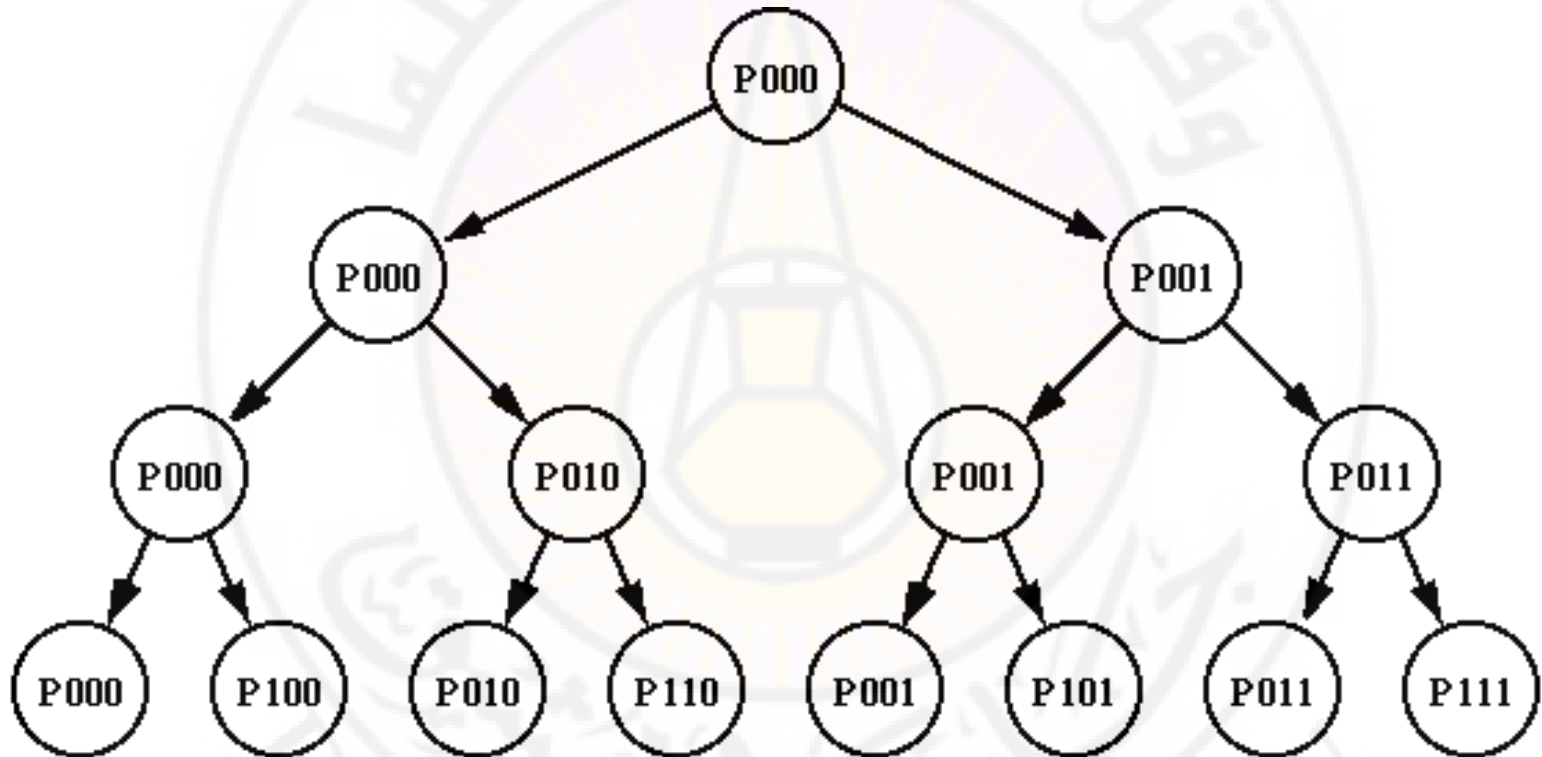


Figure 2.10 Broadcast as a tree construction

Broadcast/Gather

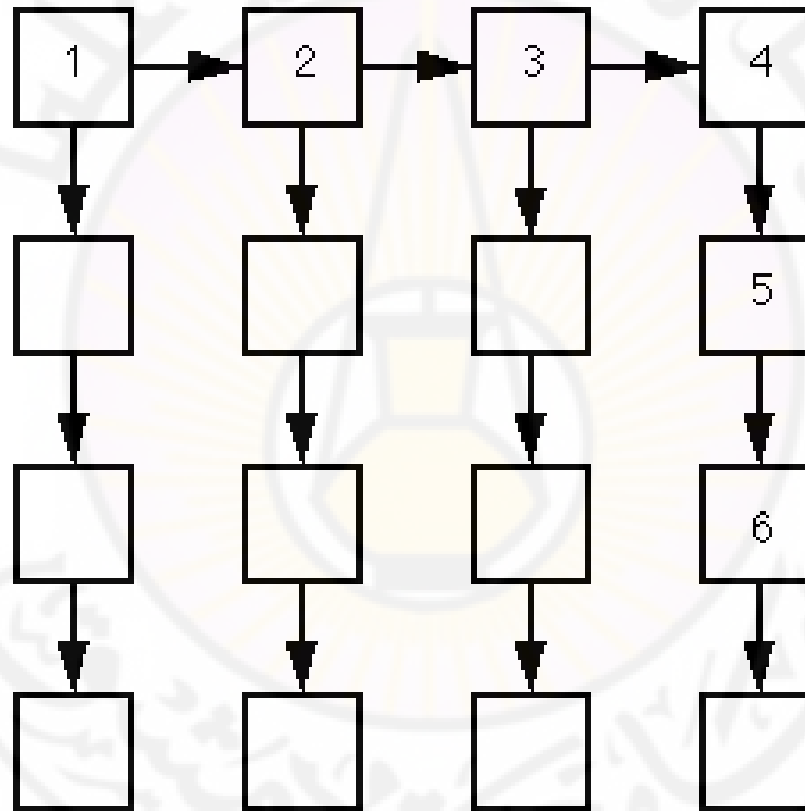
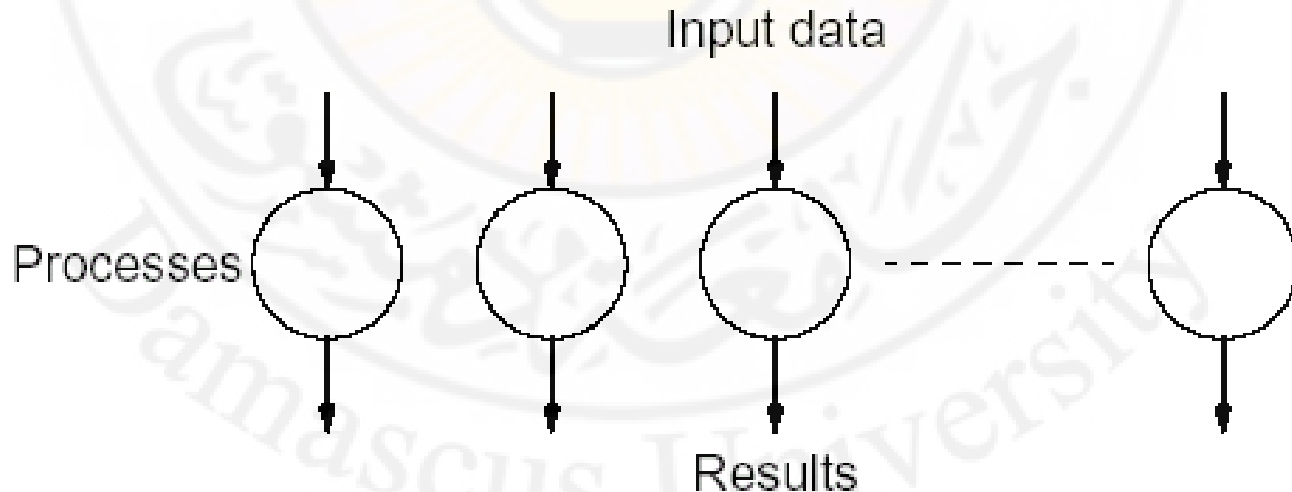


Figure 2.11 Broadcast in a mesh

الحسابات التفرعية الصريحة

Embarrassingly Parallel Computations

- يمكن لبعض الحسابات أن تقسم بشكل طبيعي إلى عدد من الأجزاء المستقلة عن بعضها البعض. ويمكن بالتالي تنفيذ هذه الأجزاء على معالجات مستقلة أو على شكل إجراءات منفصلة.
- لا يوجد اتصال بين هذه الإجراءات (أو يوجد كمية قليلة منها).
- يقوم كل إجراء بعمله دون الحاجة إلى التفاعل مع بقية الإجراءات.



الحسابات التفرعية الصريحة

Embarassingly Parallel Computations

- يؤدي ذلك إلى الحصول على أكبر قيمة ممكنة للاستجابة Speedup في حالة تخصيص إجراء لكل معالج متوفرة فترة التنفيذ.
- يجب الانتباه فقط إلى توزيع المعطيات عند بدء الاجراءات.
- يشبه هذا الاسلوب اسلوب العمل SPMD حيث لا توجد مشاركة بالمعطيات وتتوفر ذاكرة موزعة وإمكانية اتصال بواسطة الرسائل.
- في حال التعامل مع نفس المعطيات، يجب نسخ المعطيات إذن عدة مرات.
- تتطلب الحسابات التفرعية الصريحة توزيع وتجميع النتائج بطريقة ما. يمكن أن يجري ذلك من خلال إجراء يعمل لوحده في البداية وفي النهاية.

اسلوب إنشاء الإجراءات الثابت وطريقة الزبون/المخدم

Static process creation and master-slave approach



MPI approach

اسلوب إنشاء الإجراءات الثابت وطريقة الزبون/المخدم

Static process creation and master-slave approach

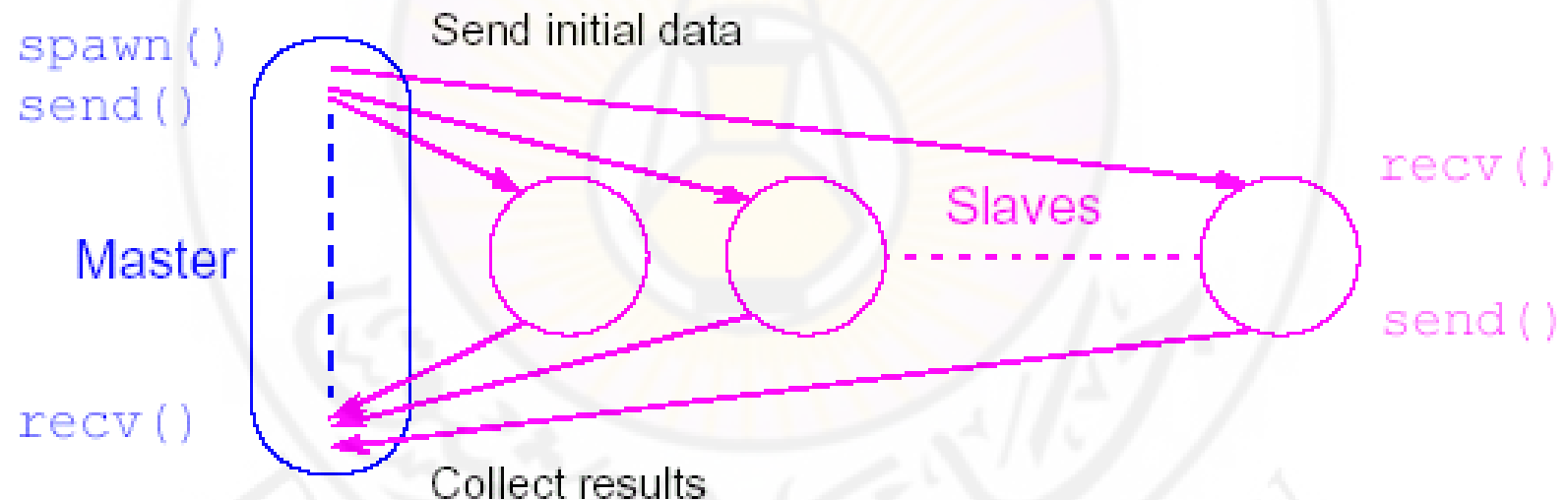
ملاحظات:

- حتى لو كانت الإجراءات متماثلة فإن التوزيع الثابت لها على المعالجات لا يؤدي بالضرورة إلى الحصول على الحل الأمثل.
- تنتج الحالة السابقة عندما تكون المعالجات غير متماثلة أو عند استخدام شبكة من محطات العمل.
- يمكن في الحالة السابقة أن نطبق أساليب موازنة الحمل Load Balancing.

اسلوب إنشاء الإجراءات الديناميكي وطريقة الزبون/المخدم

Dynamic process creation and master-slave approach

Start Master initially



PVM approach

أمثلة عن الحسابات التفرعية الصريحة

Embarrassingly Parallel Computation examples

□ التحويلات الهندسية على الصور Low level image processing

□ مجموعة ماندلبروت Mandelbrot Set.

التحويلات الهندسية على الصور

Low level image processing

تتميز عمليات التحويل في كونها تعتمد على معطيات محلية وعلى عدد محدود من عمليات الاتصال بين المناطق التي تجري عليها التحويلات.

الصورة = مصفوفة من بعدين.

حالة صورة أبيض وأسود: 1 pixel = 1 bit (١ = أبيض و ٠ = أسود)

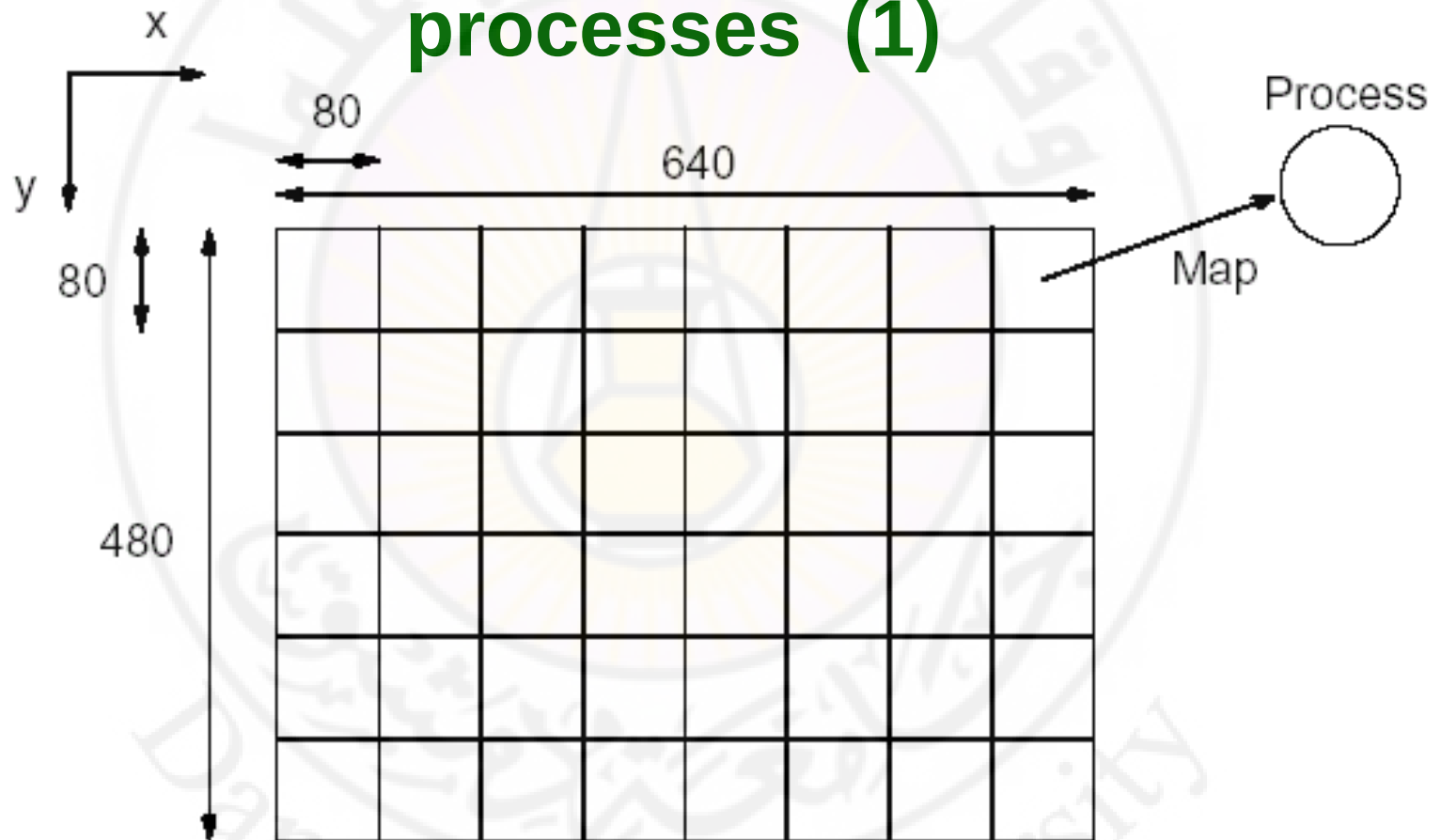
حالة صورة grayscale: 1 pixel = أكثر من bit (يمكن استخدام 8 bit لتمثيل ٢٥٦ تدرج لوني).

حالة صورة ملونة: نحتاج مثلاً إلى 3 Bytes لتحديد النسب المستخدمة من .RGB

يمكن التقليل من حجم الصورة باستخدام جدول للألوان (مثال: جدول lookup table مؤلف من ٢٥٦ لون). ونحتاج لتحديد لون كل pixel إلى 8 bits لتمثيل اللون المرتبط بهذه الـ pixel من الجدول السابق.

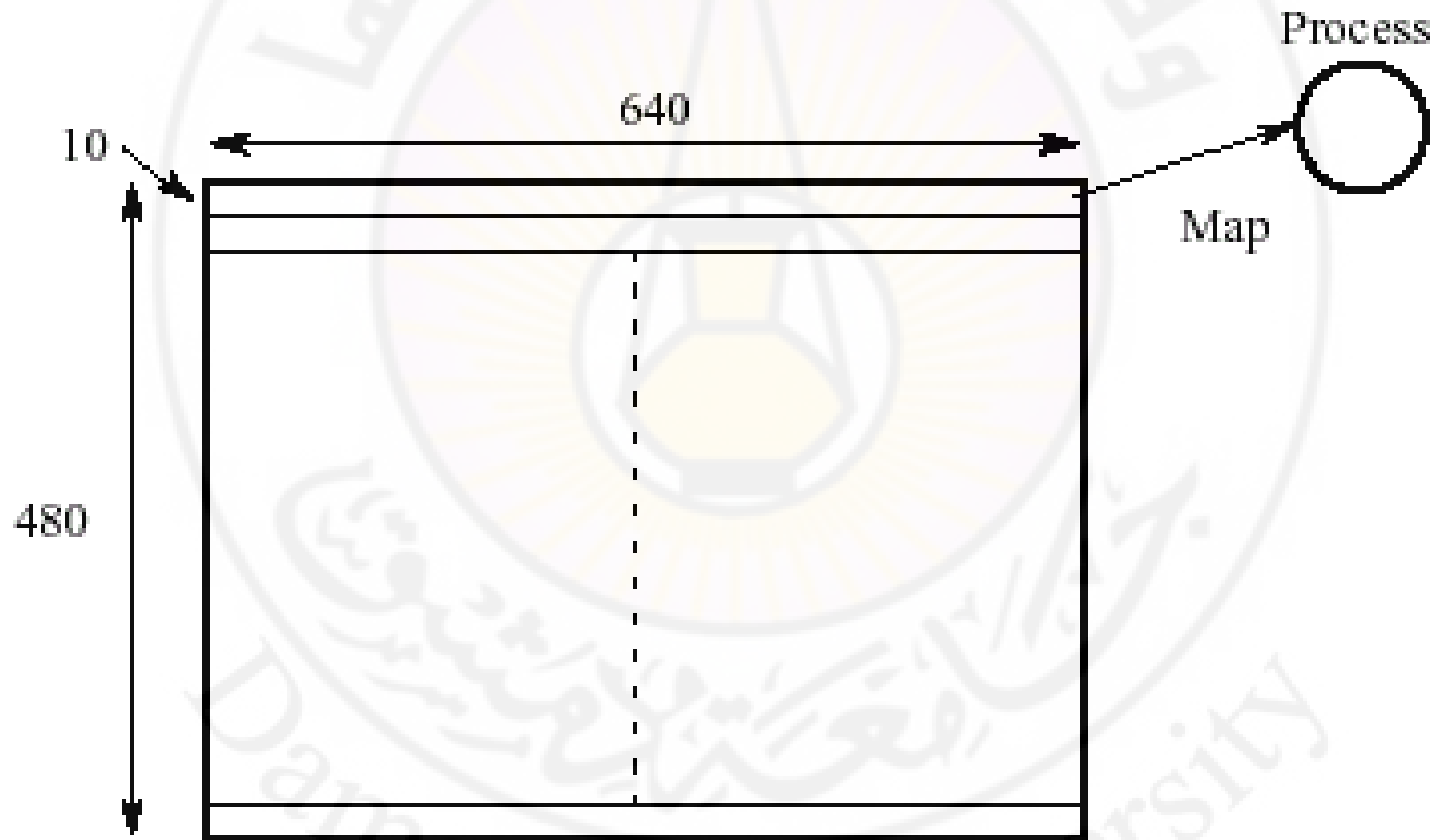
التقسيم إلى مناطق مرتبطة بالإجراءات (١)

Partitioning into regions for individual processes (1)



Square region for each process (can also use strips)

التقسيم إلى مناطق مرتبطة بالإجراءات (٢) Partitioning into regions for individual processes (2)



(b) Row region for each process

بعض عمليات التحويل الهندسي

Some geometrical operations

□ الإزاحة Shifting:

$$x' = x + \Delta x$$

$$y' = y + \Delta y$$

□ التكبير / التصغير Scaling:

$$x' = xS_x$$

$$y' = yS_y$$

□ الدوران Rotation:

$$x' = x \cos \theta + y \sin \theta$$

$$y' = -x \sin \theta + y \cos \theta$$

فكرة الحل

- نحتاج إلى برنامج Master واحد وإلى ٤٨ برنامج Slaves إذا أردنا توزيع الصورة على شكل مناطق عليها.
- الصورة عبارة عن مصفوفة `map[]` ولقد جرى إرسالها إلى كافة الإجراءات الزبونة وسوف نتعرض هنا لمسألة إزاحة الصورة فقط.
- ترسل أرقام الأسطر إلى كل إجراء حسب رقمه.
- تستقبل النتائج كل واحدة على حدة بدلا من استقبالها على شكل مجموعات.
- يجب توليد كافة النتائج ثم ينتهي بعدها عمل البرنامج Master.
- الخوارزمية السابقة ليست جيدة بشكل عام.

خوارزمية الزبون

Slave Algorithm

Recv(row, pMaster)

For (oldrow = row; oldrow < (row+10); oldrow++)

for (oldcol = 0; oldcol < 640; oldcol++) {

newrow = oldrow + delta_x;

newcol = oldcol + delta_y;

send(oldrow, oldcol, newrow, newcol, pMaster);

}

خوارزمية المخدم

Master Algorithm

```
For (i = 0, row = 0; i<48; i++, row = row +10)
    send(row, pi);
For (i = 0; i<480; i++)
    for (j = 0; j<640; j++)
        temp_map[i][j] = 0;
For (i = 0; i<(640*480); i++) {
    recv(oldrow, oldcol, newrow, newcol, pAny);
    if ! ( (newrow<0)           || (newrow>=480) ||
           (newcol<0)          || (newcol>=640) )
        temp_map[newrow][newcol] = map[oldrow][oldcol];
}
For (i = 0; i<480; i++)
    for (j = 0; j<640; j++)
        map[i][j] = temp_map[i][j] ;
```

تحليل الخوارزمية

□ إذا كان لدينا $n \times n$ عنصر (pixel) ضمن الصورة، فإن زمن الحساب للخوارزمية التسلسلية لها تساوي $t_s = n^2$ (أي $O(n^2)$) بفرص أن كل عنصر يتطلب خطوة حسابية واحدة.

□ بفرض أن عدد الإجراءات هو p وإننا نقوم بإرسال الأسطر لها تسلسلياً فإننا نحتاج إلى زمن للاتصال وقدره:

$$\begin{aligned} t_{comm} &= p(t_{startup} + 2 t_{data}) + 4n^2 (t_{startup} + t_{data}) \\ &= O(p + n^2) \end{aligned}$$

□ إذا جرى تقسيم الصورة إلى n^2/p وكان كل عنصر يتطلب عمليتي حساب لتنفيذ الإزاحة فإن:

$$t_{comp} = n^2/p = O(n^2/p)$$

□ من أجل p ثابتة فإن زمن التنفيذ التفرعي يصبح مساوياً لـ:

$$t_p = t_{comm} + t_{comp} = O(n^2)$$

ربما يكون من المفيد استخدام البنية Shared memory multi-processor

مجموعة ماندلبروت

Mandelbrot set

مجموعة من النقاط في المستوي العقدي Complex plane شبه الثابتة (يمكن لعددتها أن يزيد أو يتناقص ولكن لن يزيد عن حد معين) عند حسابها باستخدام التابع التكراري التالي:

$$Z_{k+1} = Z_k^2 + C$$

حيث أن:

Z_{k+1} هو التكرار رقم $k+1$ لحساب العدد العقدي $Z = a + bi$
 C هو عدد عقدي ثابت للدلالة على موضع النقطة ضمن المستوي العقدي.
القيمة الابتدائية لـ Z هي الصفر.

يجري تنفيذ التكرارات حتى تصبح طويلة العدد Z أكبر من القيمة ٢ أو أن يزيد عدد هذه التكرارات عن حد مختار سلفاً.

ت حسب طويلة العدد Z وفقاً للمعادلة التالية:

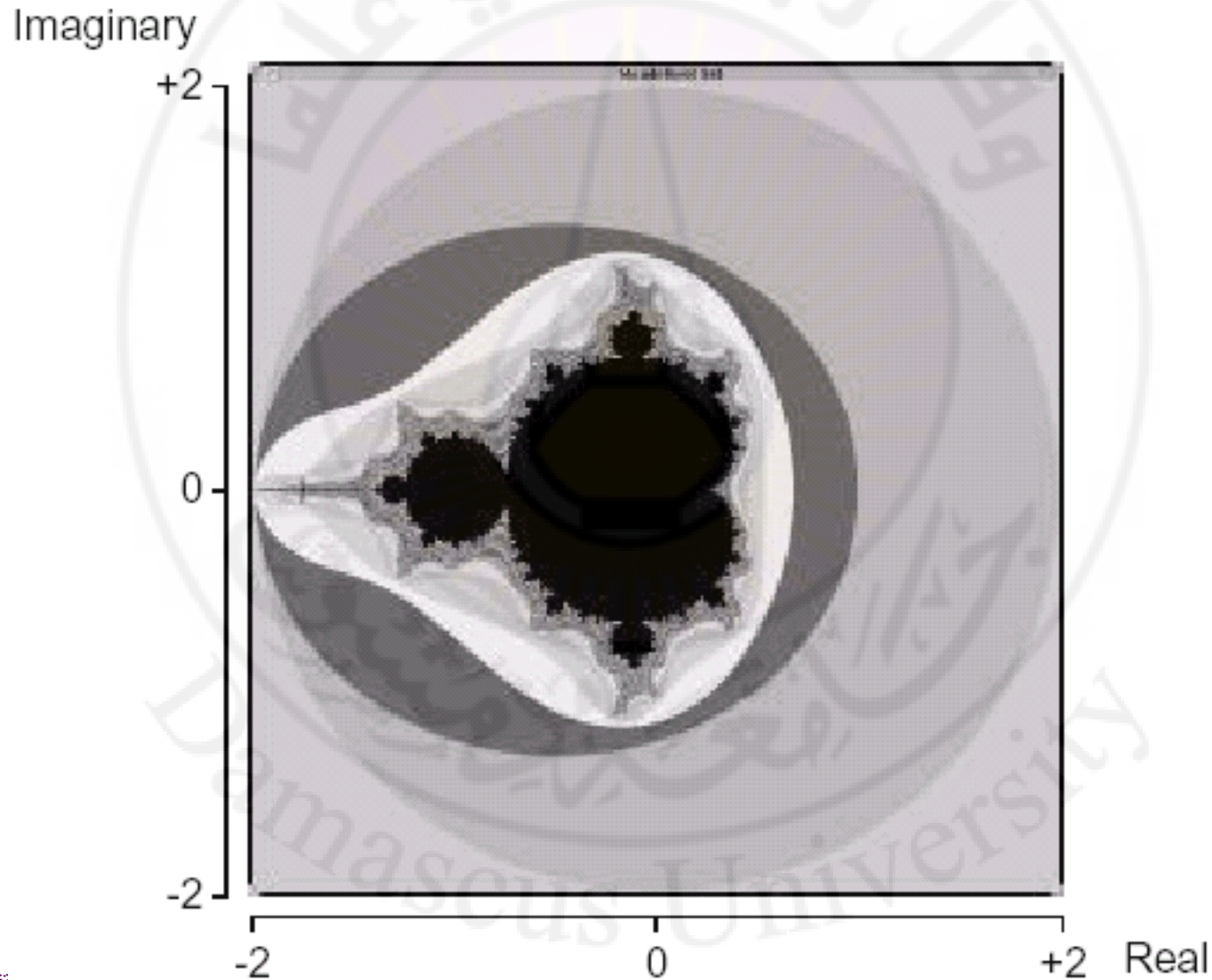
$$Z_{length} = \sqrt{a^2 + b^2}$$

البرنامج التسلسلي

Sequential routine

```
int cal_pixel(float c_real, float c_imag)
{
    int i, max;
    float z_real, z_imag, z_real_next, lengthsq;
    max = ??;
    z_real = 0; z_imag = 0;
    i = 0;                                /* number of iterations */
    DO {
        z_real_next = (z_real * z_real) - (z_imag * z_imag) + c_real;
        z_imag = (2 * z_real * z_imag) + c_imag; z_real =
        z_real_next;
        lengthsq = (z_real * z_real) + (z_imag * z_imag);
        i++;
    } WHILE (lengthsq < 4.0 AND i < max);
    return i;
}
```

مجموعة ماندلبروت Mandelbrot set



تفريع حسابات مجموعة ماندلبروت

Parallelizing Mandelbrot Set Computation

الاسناد الثابت للمهام:

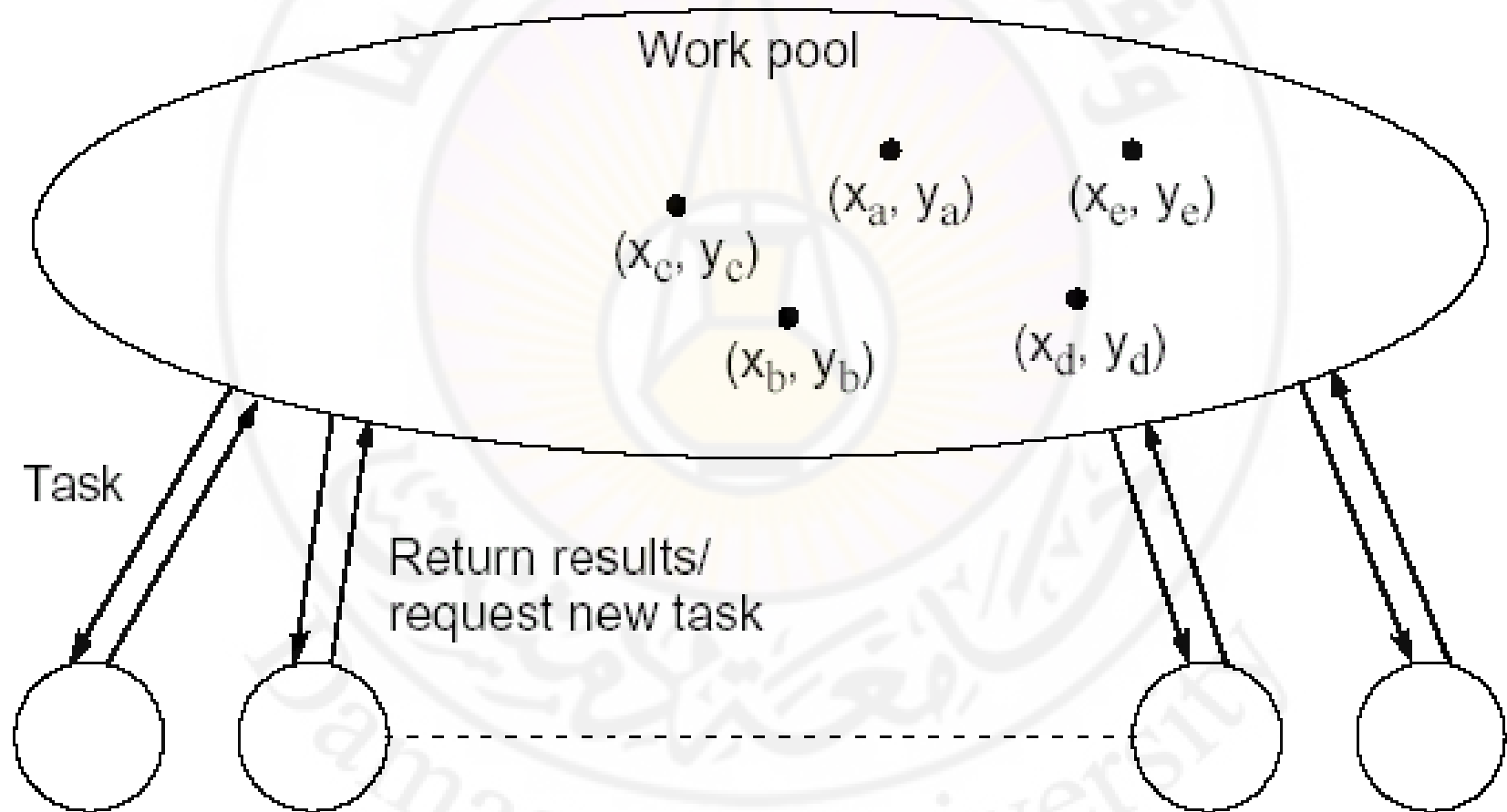
نقوم بتقسيم المنطقة المشمولة بالحسابات إلى عدد ثابت من الأجزاء ويتولى الحسابات المرتبطة بكل جزء معالج مستقل. هذه الطريقة ليست طريقة ناجحة وذلك بسبب اختلاف كل منطقة من حيث عدد الحسابات (التكرارات) والزمن.

الاسناد الديناميكي للمهام:

يقوم كل معالج بطلب مناطق أخرى بعد انتهائه من المناطق السابقة.

الاسناد الديناميكي للمهام

Dynamic Task Assignment



التجزئة وسياسات فرق تسد

Partitioning and Divide-and-Conquer Strategies

□ تهدف عملية التجزئة إلى تقسيم المسألة المراد حلها إلى أجزاء أصغر، ثم يجري تجميع نتائج الأجزاء المختلفة.

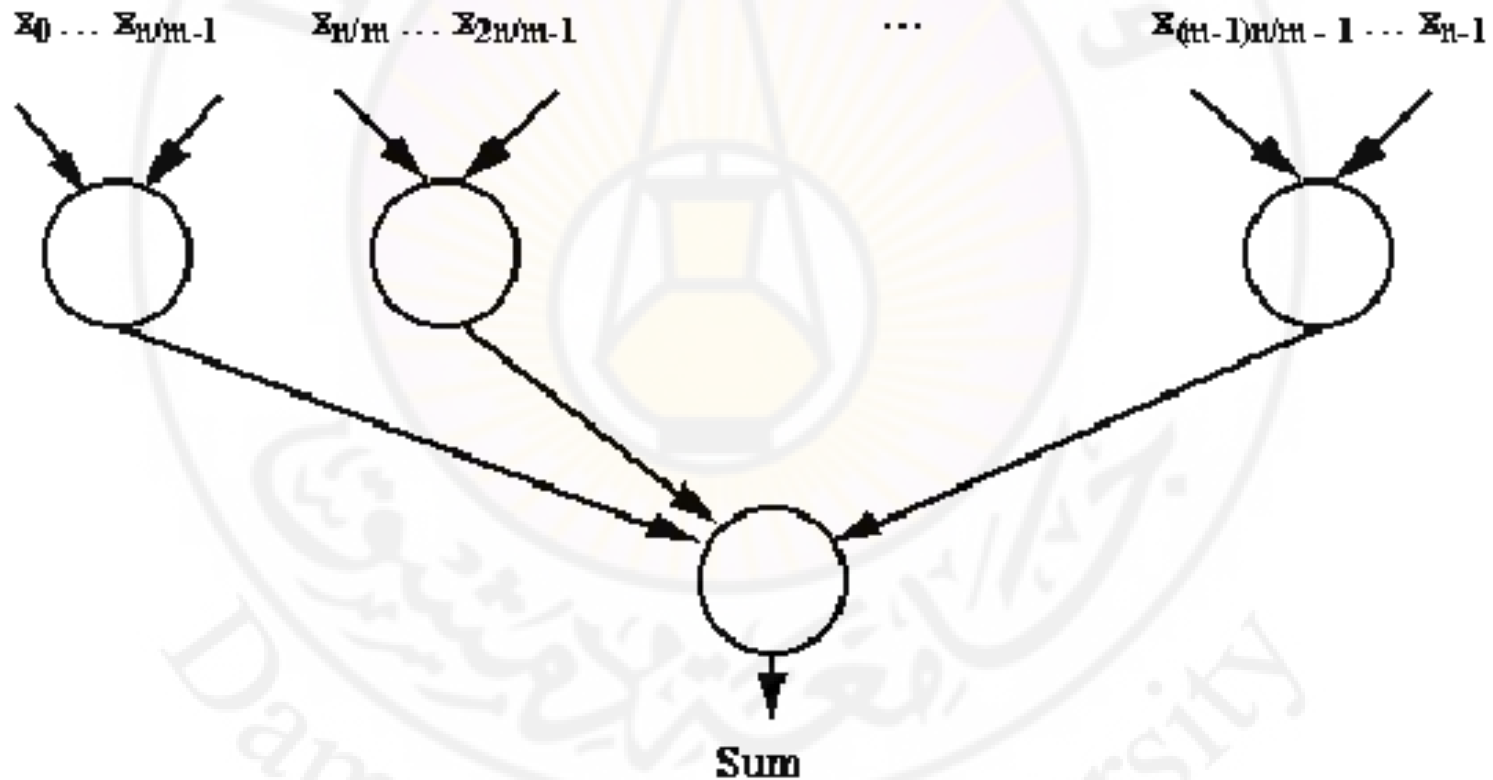
□ لدينا نوعين من أنواع التجزئة:

١. التجزئة على مستوى المعطيات data decomposition، مثال: جمع مجموعة كبيرة من الأعداد.

٢. التجزئة على مستوى الوظائف functional decomposition، وهو أمر غير بديهي في العديد من الحالات.

جمع عدة أعداد باستخدام أساليب التجزئة

Adding Numbers by partitioning



حل (١): إرسال n/m عدد

Solution (1): sending n/m numbers

□ Master:

```
s=n/m;  
For (l=0, x= 0; l<m; l++, x=x+s)  
    Send(&number[x], s, Pi);  
sum = 0;  
For (l=0; l<m; l++) {  
    recv(&part_sum, Pany);  
    sum = sum + part_sum;  
}
```

□ Slave

```
Recv(numbers, s, Pmaster);  
Part_sum =0;  
For (l=0; l<s; l++)  
    part_sum = part_sum + numbers[l];  
Send(&part_sum, Pmaster);
```

حل (2): إرسال كامل قائمة الأعداد

Solution (2): sending the complete list of numbers

□ Master:

```
s=n/m;  
bcast(numbers, s, Pslave_group);  
sum = 0;  
For (l=0; l<m; l++) {  
    recv(&part_sum, Pany);  
    sum = sum + part_sum;  
}
```

□ Slave

```
bcast(numbers, s, Pmaster);  
Start = slave_number * s;  
End = start + s;  
part_sum = 0;  
For (l=Start; l<End; l++)  
    part_sum = part_sum + numbers[l];  
Send(&part_sum, Pmaster);
```


حل (٣): استخدام Reduce و Scatter

□ Master:

$s = n/m;$

Scatter (numbers, &s, Pgroup, root=master);

Reduce_add (&sum, &s, Pgroup, root=master);

□ Slave

Scatter (numbers, &s, Pgroup, root=master);

Reduce_add (&sum, &s, Pgroup, root=master);

جمع عدة أعداد باستخدام أساليب التجزئة

Adding Numbers by Partitioning

- نحتاج إلى حساب عدد العمليات الحسابية. يقوم m معالج بجمع n/m عدد، أي يقوم كل منها بتنفيذ $n/m-1$ عملية حسابية.
- نحتاج لتنفيذ عملية تجميع نتائج المجاميع الجزئية إلى تنفيذ $m-1$ عملية حسابية إضافية.
- نحصل بالتالي على زمن حساب تفرعي مساوي للقيمة التالية:

$$t_{\text{comp}} = n/m - 1 + m - 1$$

$$t_{\text{comp}} = O(n/m + m)$$

$$t_p = n + n/m + m - 2 = O(n + n/m + m)$$

- بفرض أنه تم إرسال كل عدد على حدى.
- أما بالنسبة لزمن الحساب التسلسلي فيساوي:

$$t_s = n - 1 = O(n).$$

جمع عدة أعداد باستخدام أساليب التجزئة

Adding Numbers by Partitioning

□ تفرعياً تمر عمليات الحساب بعدة مراحل.

□ المرحلة الأولى - اتصال لتمرير الأعداد المخصصة لكل إجراء زيون:

□ في حالة استخدام توابع إرسال واستقبال فردية:

$$t_{\text{comm1}} = m(t_{\text{startup}} + (n/m)t_{\text{data}})$$

□ في حالة استخدام توابع scatter و reduce:

$$t_{\text{comm1}} = t_{\text{startup}} + n t_{\text{data}}$$

□ المرحلة الثانية - حسابات لجمع n/m عدد:

$$t_{\text{comp1}} = n/m - 1$$

□ المرحلة الثالثة - اتصال لتجميع النتائج:

□ في حالة استخدام توابع إرسال واستقبال فردية:

$$t_{\text{comm2}} = m(t_{\text{startup}} + t_{\text{data}})$$

□ في حالة استخدام توابع scatter و reduce:

$$t_{\text{comm2}} = t_{\text{startup}} + m t_{\text{data}}$$

□ المرحلة الرابعة - حسابات لتجميع النتائج:

$$t_{\text{comp2}} = m - 1$$

أساليب فرق تسد

Divide-and-Conquer Strategies

- تهدف هذه الأساليب إلى تقسيم المسألة المطروحة إلى مسائل أصغر تكرارياً Recursion.
- عندما يجري التقسيم ثنائياً، نحصل على شجرة ثنائية. ونغطي الشجرة نزولاً عند التقسيم وصعوداً عند إعادة النتائج.
- يمكن أن تجري عملية التقسيم إلى أكثر من مسألتين جزئيتين.

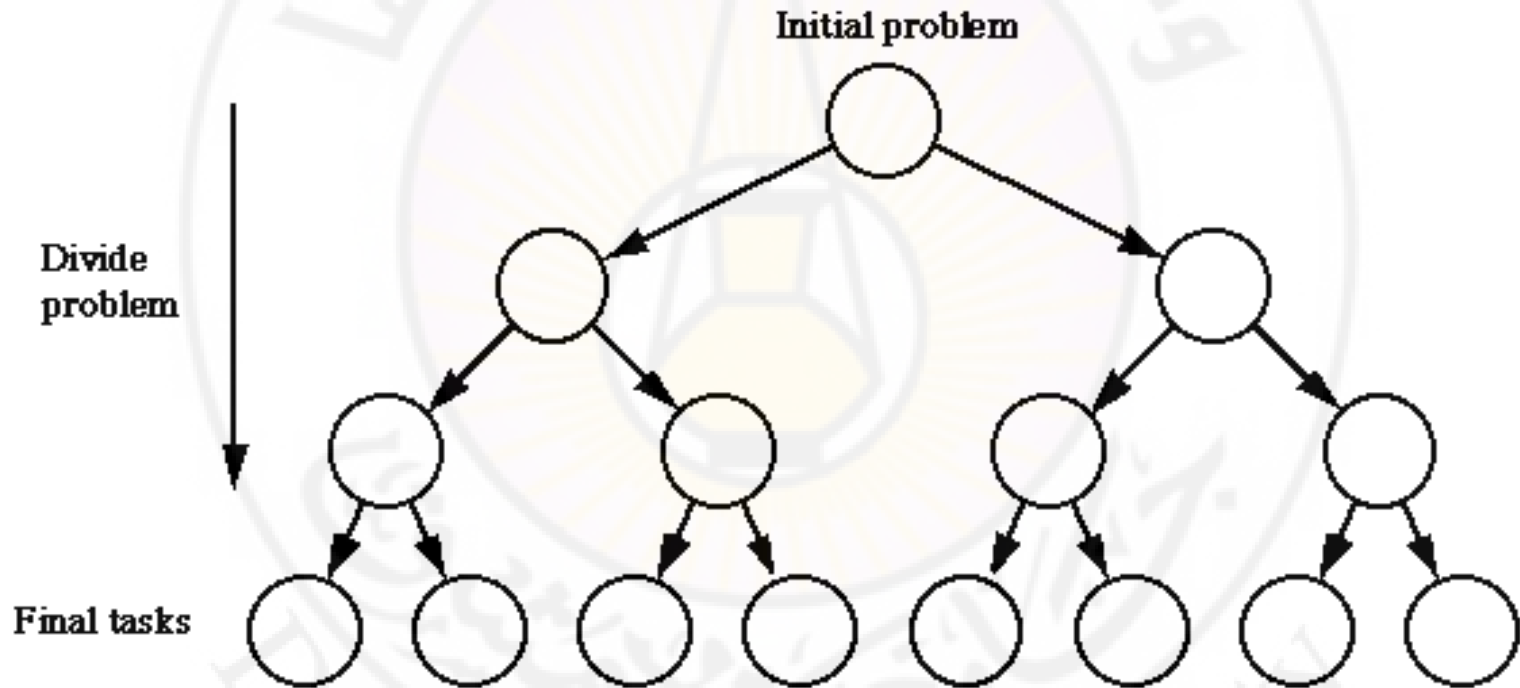
تعريف العودية التسلسلية

□ مثال:

```
Int add(int *s)
{
    if (number(s) =< 2) return (n1+ n2);
    else {
        divide(s, s1, s2);
        part_sum1 = add(s1);
        part_sum2 = add(s2);
        return (part_sum1 + part_sum2);
    }
}
```

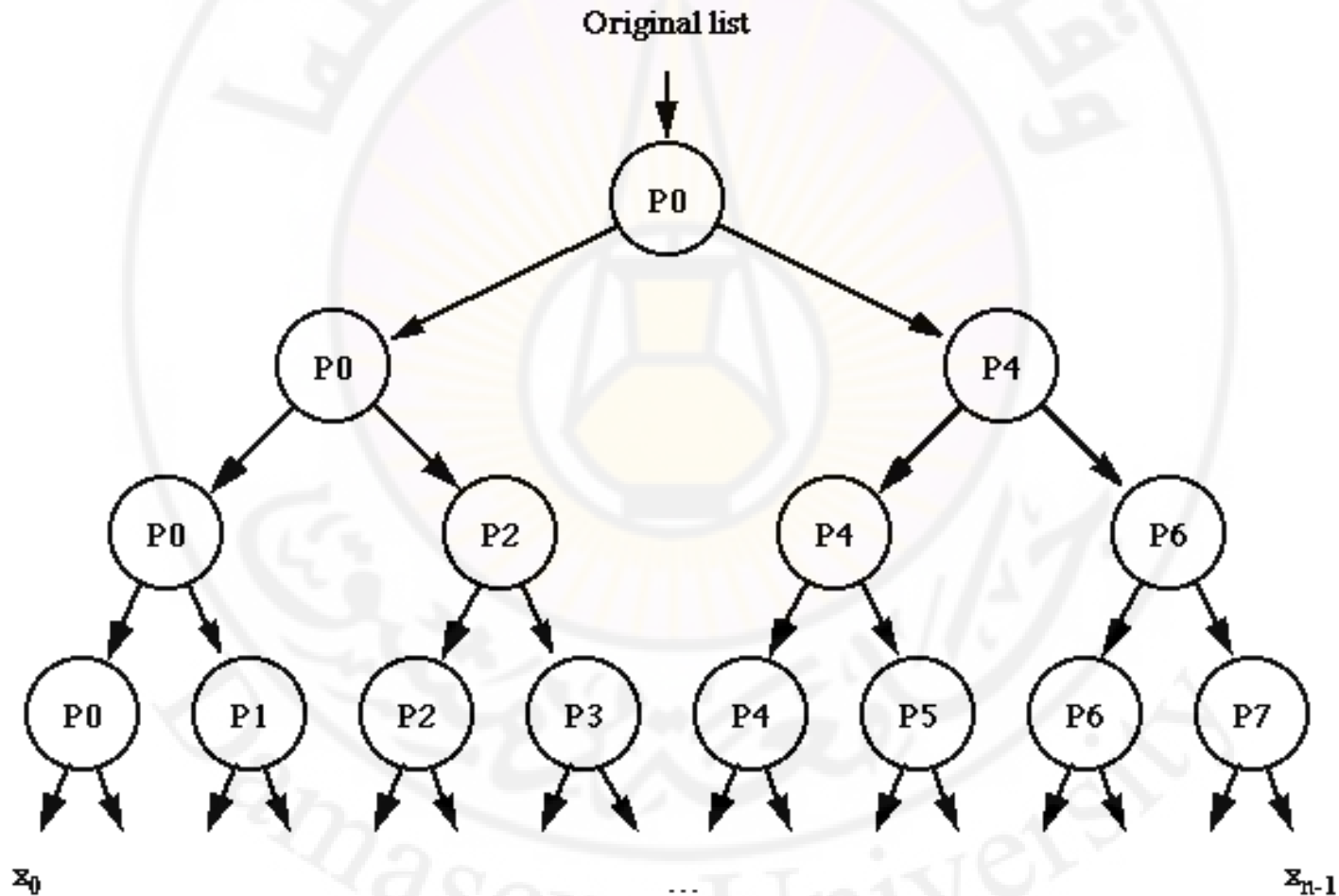
تقسيم المسألة

Dividing a Problem

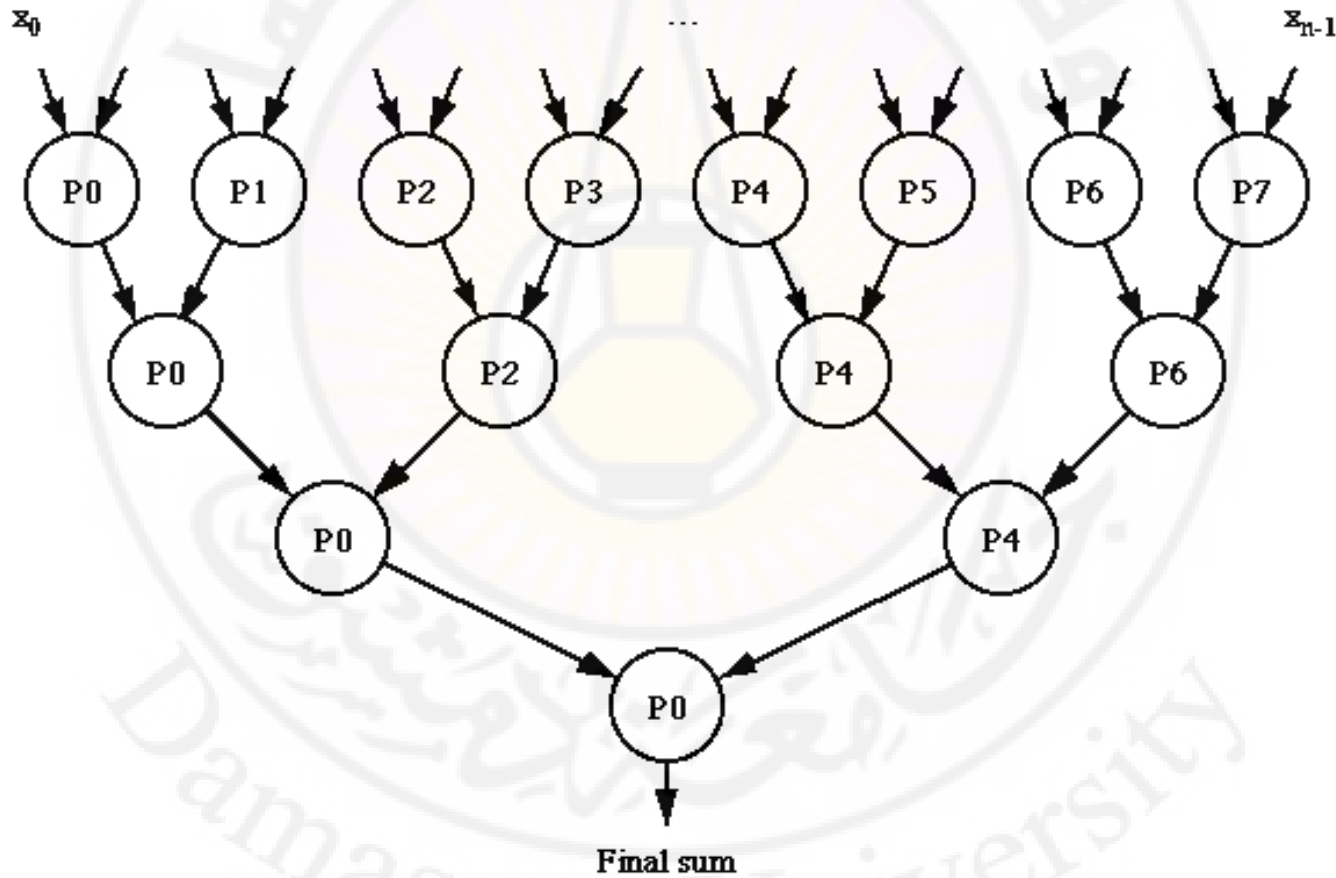


تقسيم قائمة إلى أجزاء

Dividing a List into Parts



تجميع النتائج Partial Summation



Process P_0

□ Process P_0

```
Divide(s1, s1, s2);  
Send(s2, P4);  
Divide(s1, s1, s2);  
Send(s2, P2);  
Divide(s1, s1, s2);  
Send(s2, P1);  
Part_sum = *s1;  
Recv(&part_sum1, P1);  
Part_sum = Part_sum + part_sum1;  
Recv(&part_sum1, P2);  
Part_sum = Part_sum + part_sum1;  
Recv(&part_sum1, P4);  
Part_sum = Part_sum + part_sum1;
```

Process P_4

□ Process P_4

```
Recv(s1, P0);  
Divide(s1, s1, s2);  
Send(s2, P6);  
Divide(s1, s1, s2);  
Send(s2, P5);  
Part_sum = *s1;  
Recv(&part_sum1, P5);  
Part_sum = Part_sum + part_sum1;  
Recv(&part_sum1, P6);  
Part_sum = Part_sum + part_sum1;  
Send(&part_sum1, P0);
```

تحليل الخوارزمية

□ بفرض أن $n = 2^x$ وأن $t_{startup}$ مهمل، فإن كلفة الاتصال تصبح:

$$t_{comm1} = n/2 t_{data} + n/4 t_{data} + \dots + n/p t_{data} \\ = n(p-1)/p t_{data}$$

$$t_{comm2} = \log p t_{data}$$

$$t_{comm} = t_{comm1} + t_{comm2} = n(p-1)/p t_{data} + \log p t_{data} = O(n) \\ \text{for const. } p$$

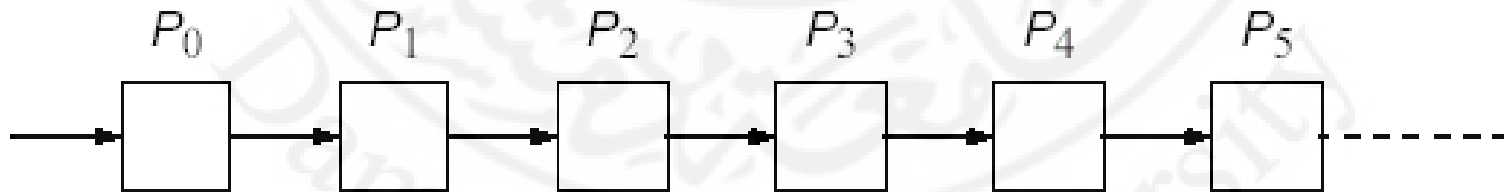
□ أما بالنسبة للحسابات فلدينا الكلفة التالية:

$$t_{comp} = n/p + \log p$$

الحسابات باستخدام الـ Pipeline

Pipelined Computations

- تهدف إلى تجزئة مسألة ما إلى سلسلة من المهام التي يجب تنفيذها واحدة تلو الأخرى. يمكن تنفيذ كل مهمة باستخدام إجراء أو معالج مستقل.
- تشبه طريقة التقسيم السابقة، طريقة التقسيم الوظيفي functional decomposition مع إضافة فكرة تنظيم وتنفيذ المهام تسلسلياً وراء بعضها البعض.
- تسمى كل مرحلة من مراحل الـ Pipeline بـ Pipeline Stage.



مثال Example

حساب مجموع عناصر مصفوفة a .

```
For (i=0; i<n; i++)  
    sum = sum + a[i];
```

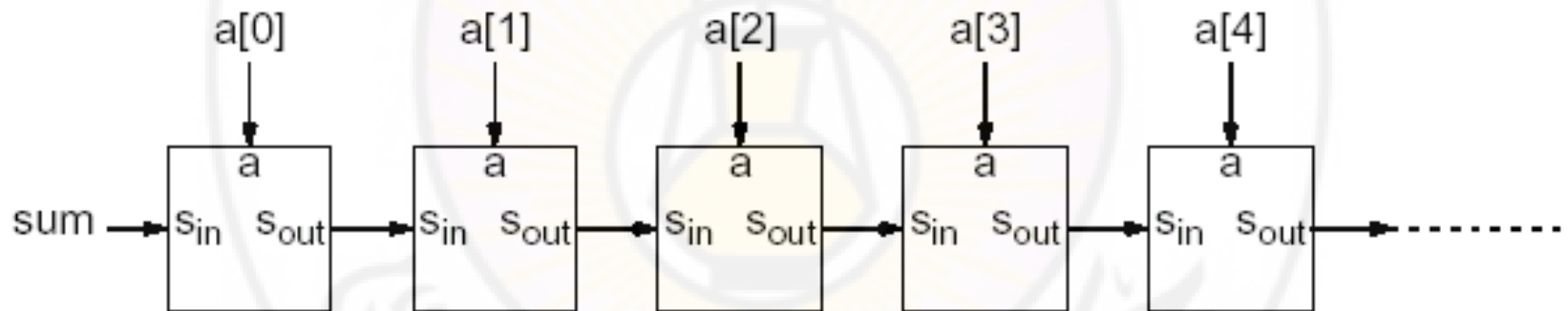
يمكن كتابة الحلقة السابقة على الشكل التالي:

```
sum = sum + a[0];  
sum = sum + a[1];  
sum = sum + a[2];  
sum = sum + a[3];
```

...

يمكن التعبير عن التعليمات السابقة على شكل مرحلة من مراحل Pipeline

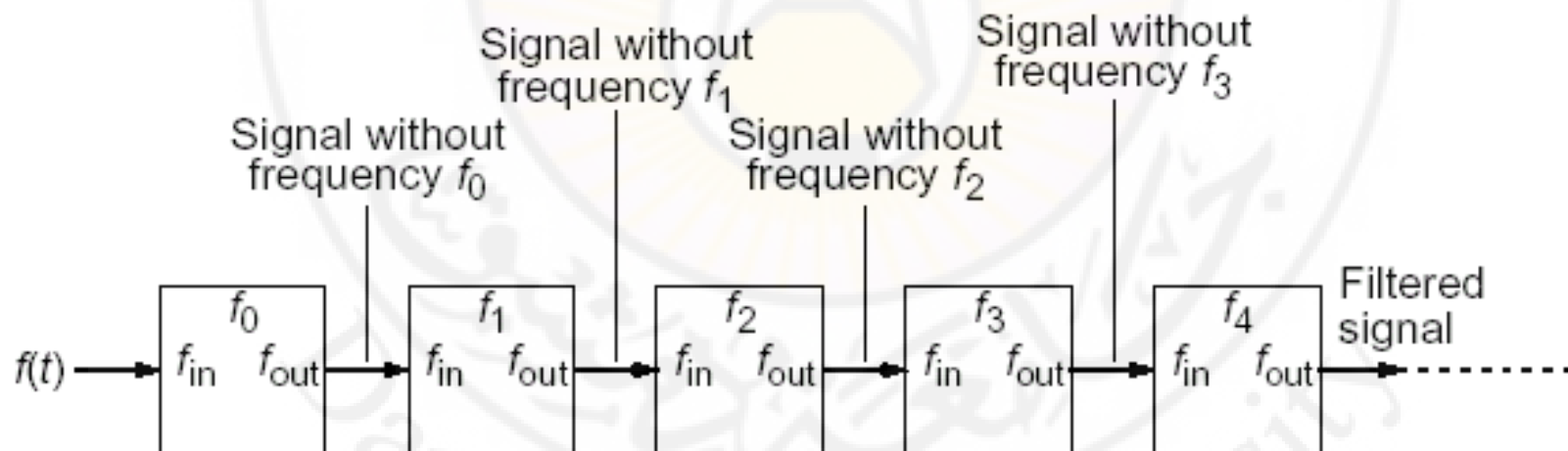
مثال Example



مثال آخر

Another Example

- فلترة الترددات Frequency Filter من أجل حذف مجموعة من الترددات (f_0, f_1, f_2 , etc) من إشارة ما $f(t)$.
- تدخل الإشارات إلى ال Pipeline من اليسار.



أين يمكن استخدام تقنية الـ Pipeline؟

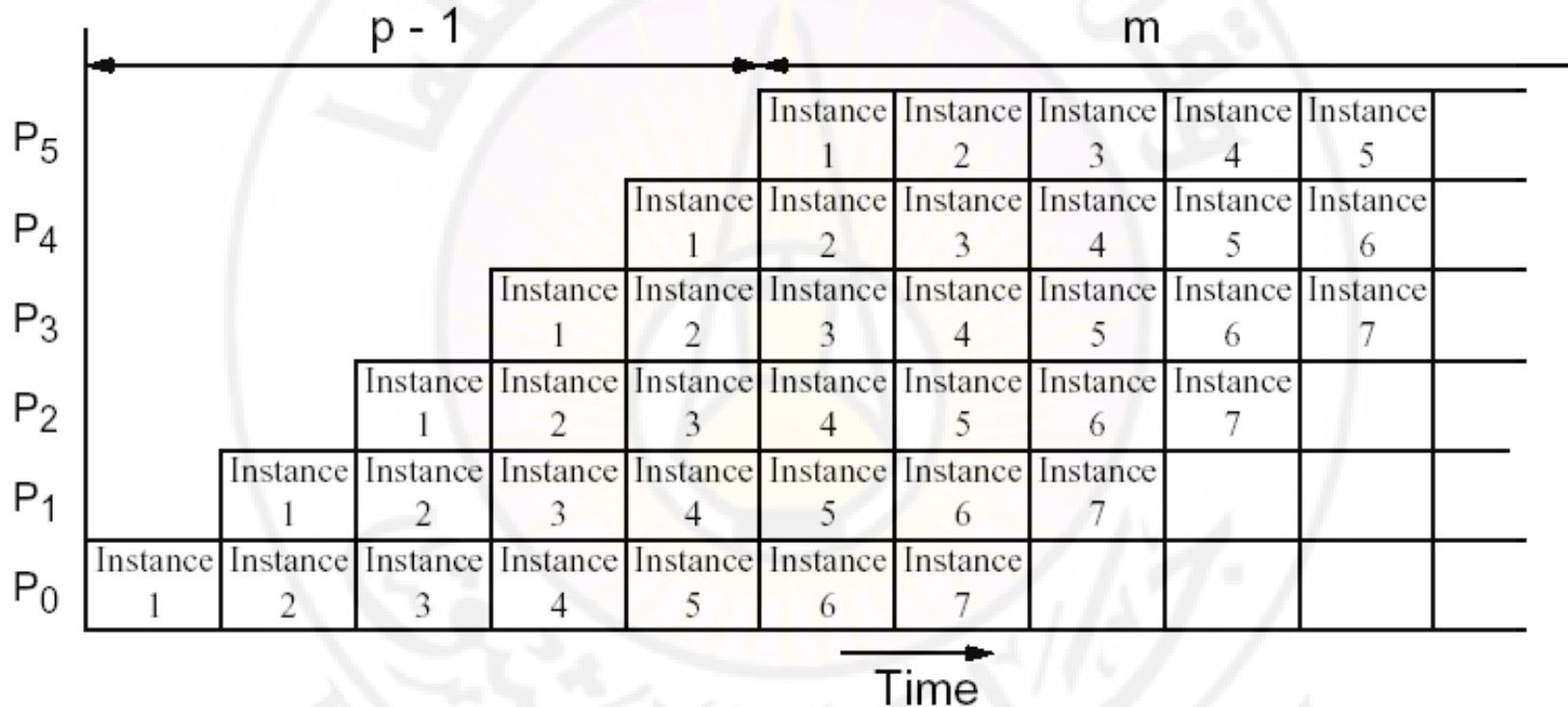
Where pipelining can be used?

□ تساهم هذه التقنية في تحسين الأداء وفي تسريع التنفيذ بالنسبة لأنواع الحسابات التالية:

١. إذا توفر لدينا أكثر من نسخة كاملة عن المسألة (type 1).
٢. إذا توفر لدينا سلاسل من المعطيات يجب معالجتها وهي تتطلب تنفيذ عدة عمليات (type 2).
٣. إذا كانت المعلومات اللازمة لإطلاق عملية المعالجة التالية لا تستدعي إنتهاء كافة أعمال مرحلة المعالجة السابقة (type 3).

المخطط الزمني لمسائل النوع الأول

“Type 1” Pipeline Space-Time Diagram

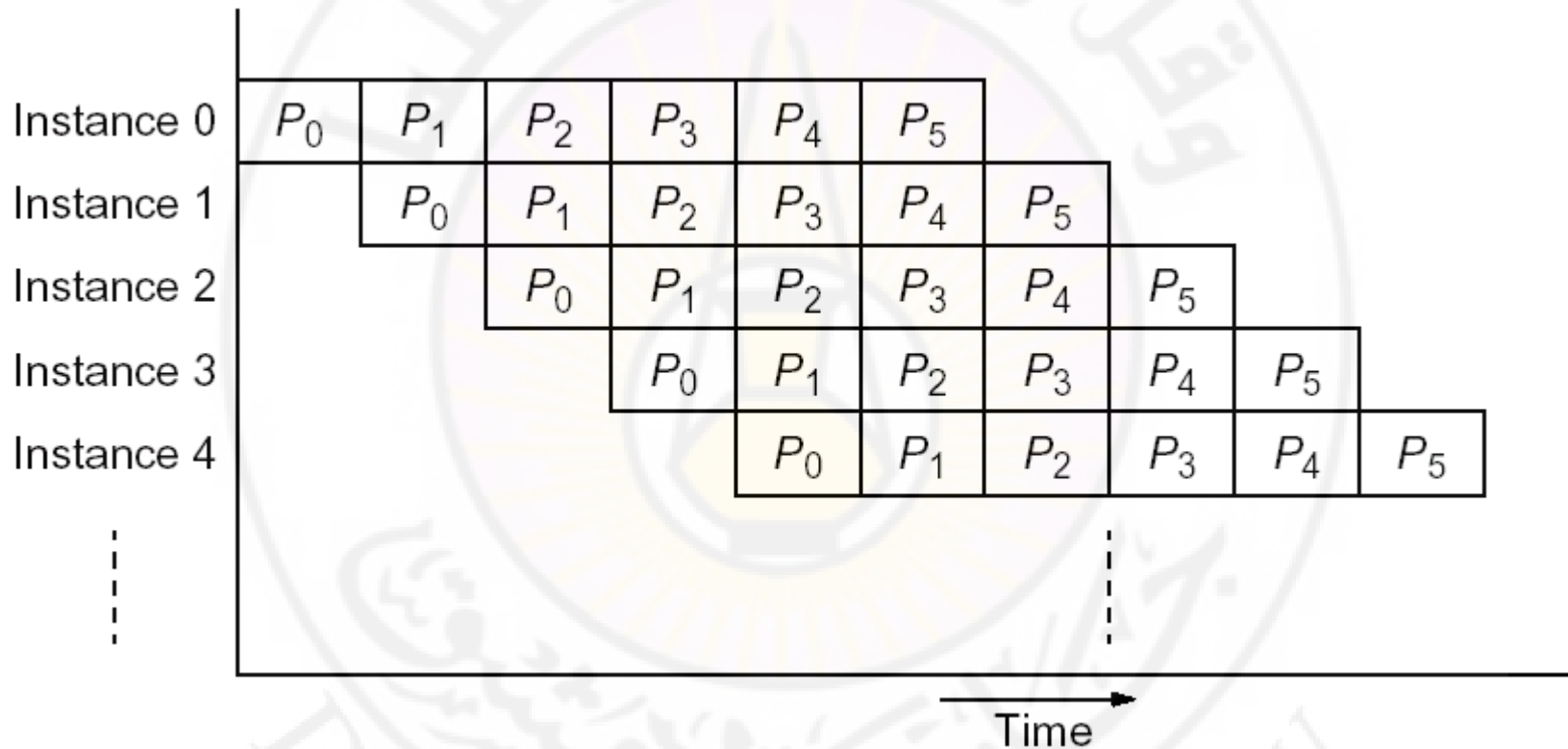


زمن التنفيذ = $m + p - 1$ حلقة من أجل p مرحلة و m نسخة.

وسطي زمن التنفيذ = $(m + p - 1) / m$ وهي قيمة تقترب من 1 عندما تسعي m إلى اللانهاية. هذا يعني أن هناك إتمام لمسألة في كل حلقة.

المخطط الزمني لمسائل النوع الأول

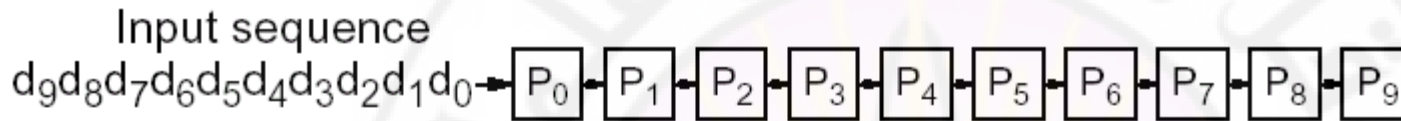
“Type 1” Pipeline Space-Time Diagram



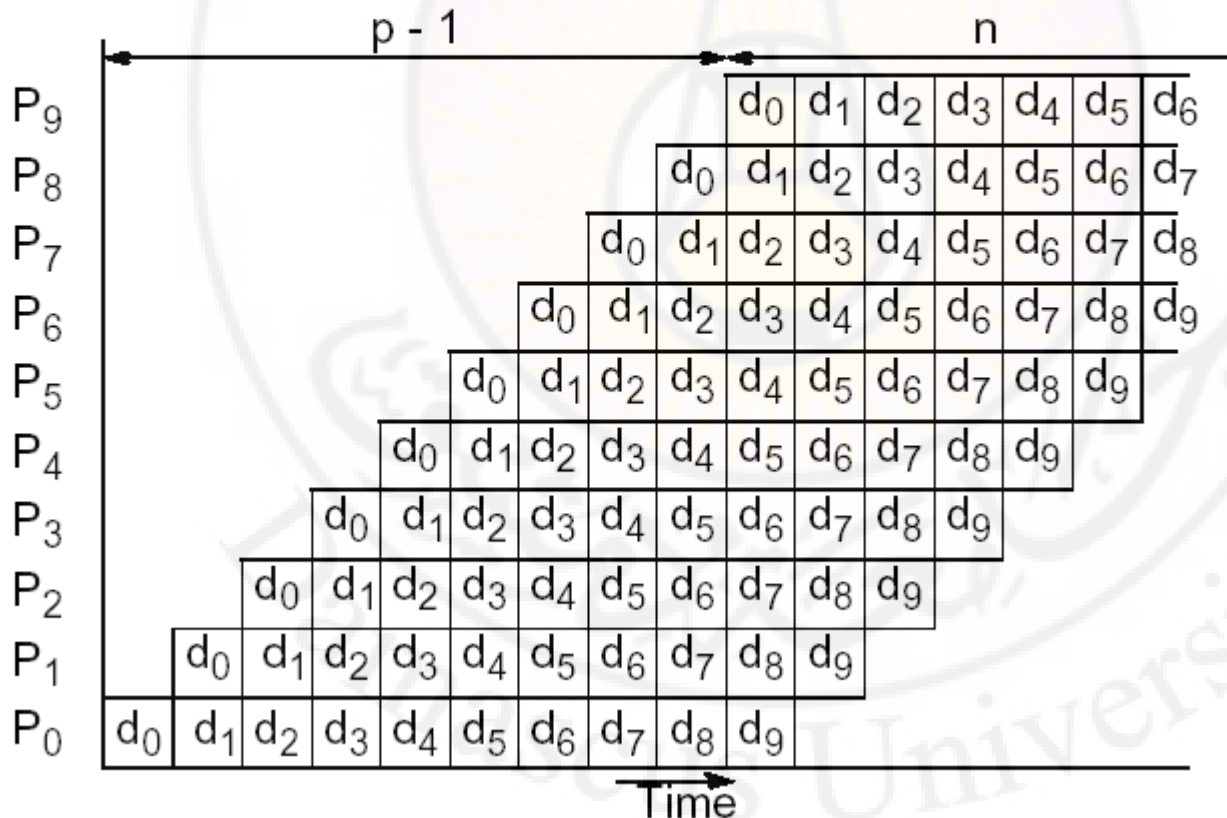
يمكن التعبير عن معلومات الشكل السابق بطريقة أخرى كما هو مبين في الشكل المذكور أعلاه.

المخطط الزمني لمسائل النوع الثاني

“Type 2” Pipeline Space-Time Diagram



(a) Pipeline structure

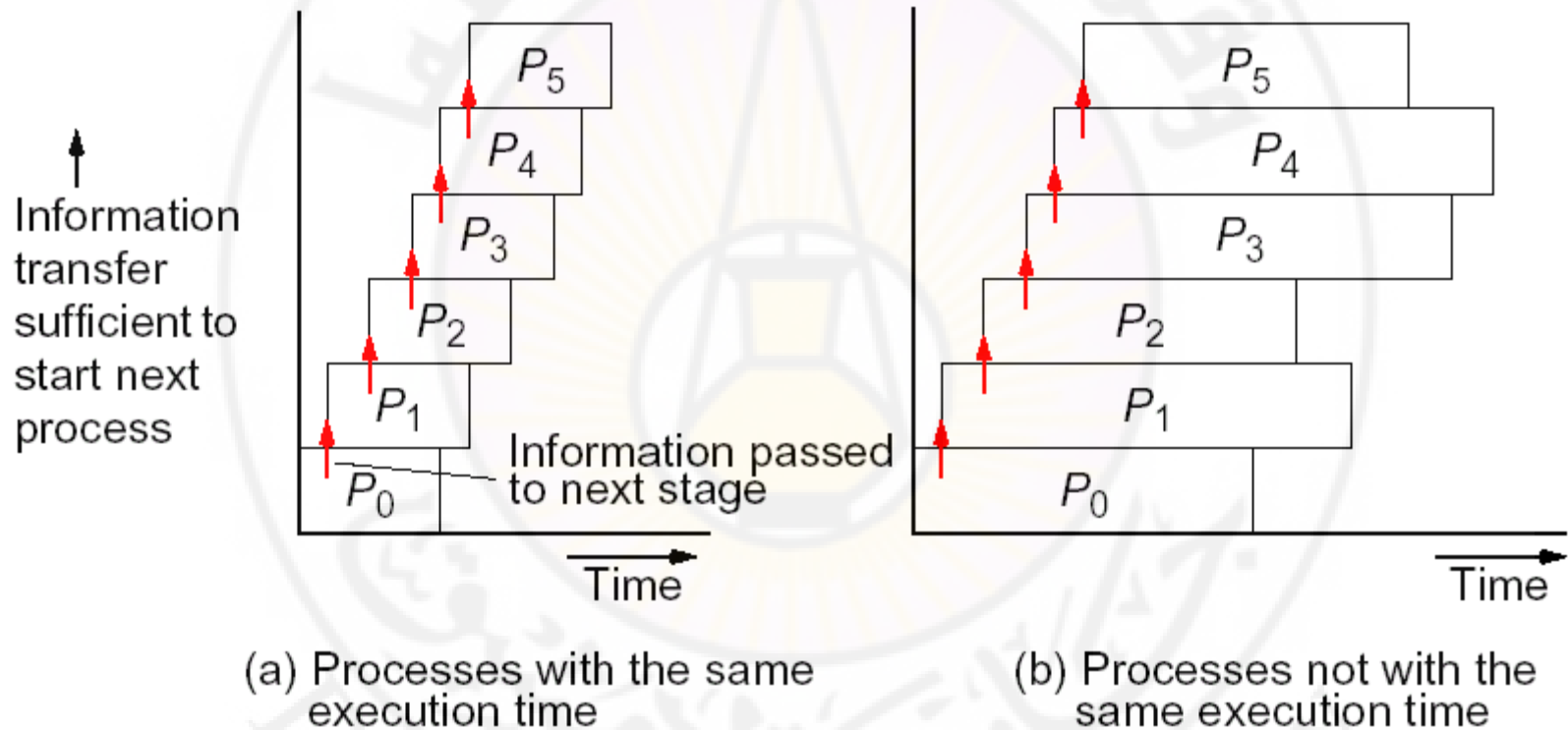


إجمالي زمن التنفيذ
 $n + (p - 1) =$

حيث n هو عدد
 معطيات الدخل

المخطط الزمني لمسائل النوع الثالث

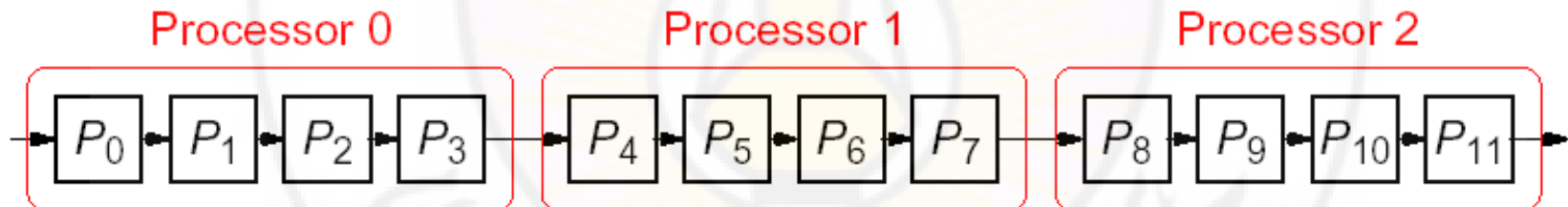
“Type 3” Pipeline Space-Time Diagram



يجري تمرير المعلومات إلى المرحلة اللاحقة قبل نهاية تنفيذ هذه المرحلة.

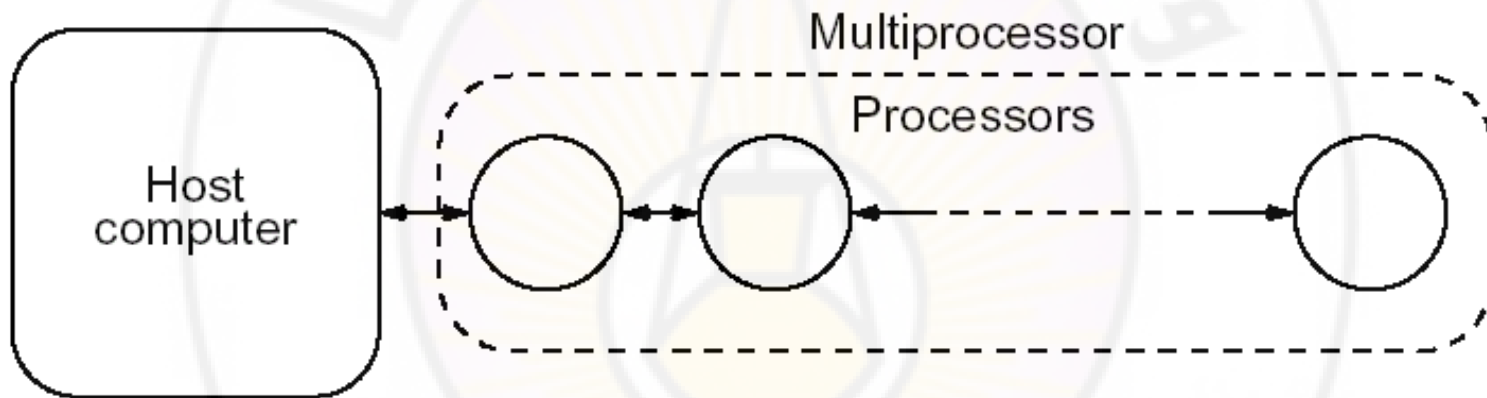
ملاحظة Note

إذا كان عدد المراحل أكبر من عدد المعالجات المتواجدة في الـ Pipeline، يجب عندها إسناد عدة مراحل إلى كل معالج.



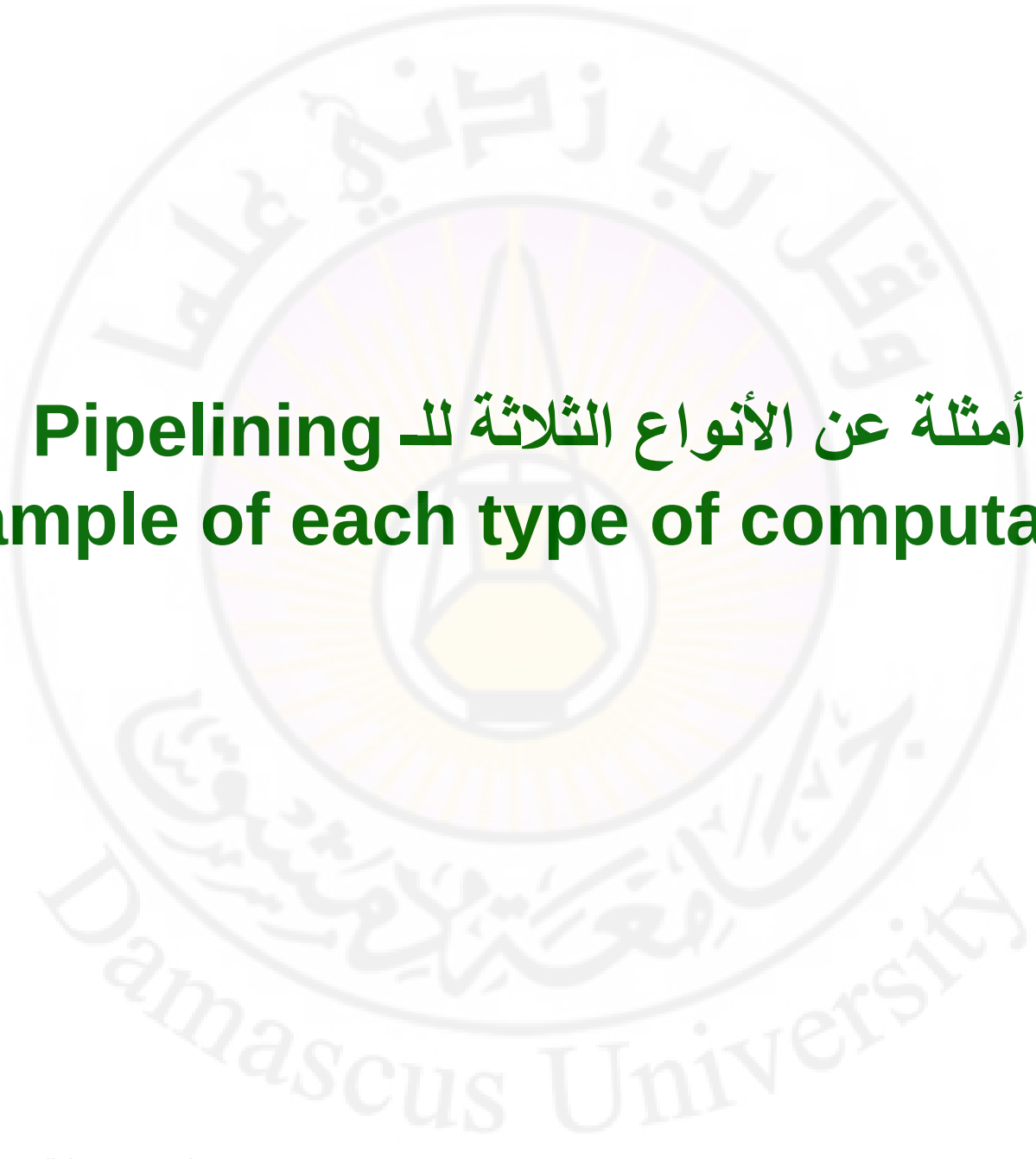
منصة تنفيذ تطبيقات الـ Pipeline

Computing Platform for Pipelined Applications



تعتبر متعددات المعالجات ذا طريقة الربط الخطية Line configuration من أفضل الطرق إضافة إلى طريقة الربط الحلقية Ring configuration.

يجب استخدام تعليمات إرسال موقفة Blocking send من أجل تحقيق التسلسل المطلوب لعمليات الـ Pipeline.

The background of the slide features a large, faint watermark of the Damascus University logo. The logo is circular, with a central emblem depicting a stylized building or monument. The emblem is surrounded by a ring of Arabic text, and the words "Damascus University" are written in English at the bottom of the circle.

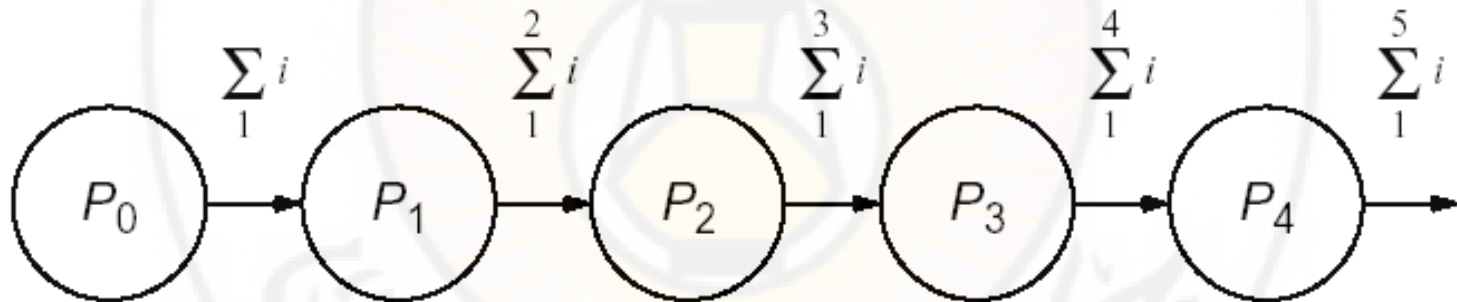
أمثلة عن الأنواع الثلاثة للـ `Pipelining`

Example of each type of computation

مثال (١): جمع الأعداد

Example (1): Adding Numbers

عملية Pipeline من النوع الأول.



مثال (١): جمع الأعداد

Example (1): Adding Numbers

البرنامج الرئيسي لأي إجراء P_i :

```
recv(&accumulation,  $P_{i-1}$ );
```

```
accumulation = accumulation + number;
```

```
send(&accumulation,  $P_{i+1}$ );
```

ما عدا الإجراء رقم صفر P_0 الذي يقوم بالعملية التالية:

```
send(&number,  $P_1$ );
```

والإجراء الأخير P_{n-1} الذي يقوم أيضاً بتنفيذ ما يلي:

```
recv(& accumulation,  $P_{n-2}$ );
```

```
accumulation = accumulation + number;
```

برنامج SPMD

SPMD Program

```
If (process > 0) {  
    recv(&accumulation,  $P_{i-1}$ );  
    accumulation = accumulation + number;  
}
```

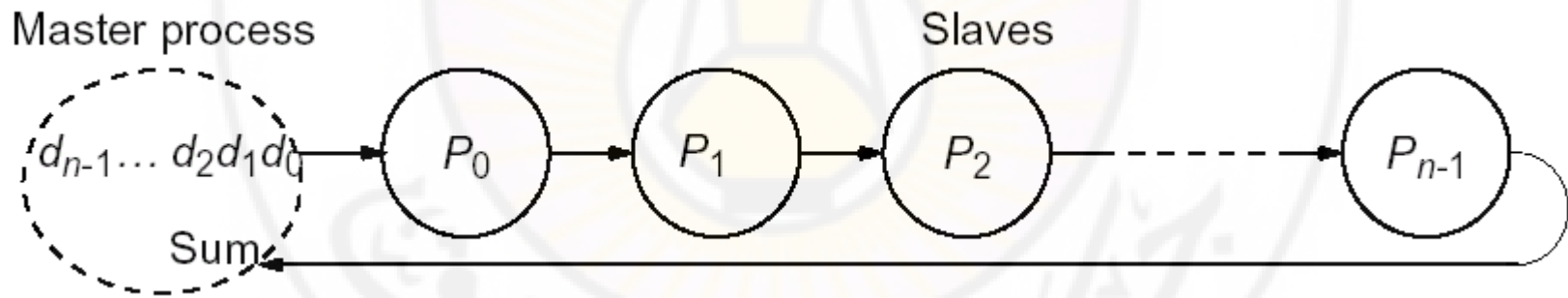
```
If (process < n-1) send(&accumulation,  $P_{i+1}$ );
```

تصل النتيجة النهائية إلى الإجراء الأخير.

يمكن استخدام نفس أسلوب المعالجة من أجل عمليات حسابية أخرى غير عملية الجمع.

جمع الأعداد بالطريقة الحلقية

Adding Numbers with Ring Configuration



تحليل مسألة من النوع الأول

$$\begin{aligned} t_{total} &= \text{عدد الحلقات} * \text{زمن تنفيذ حلقة واحدة} \\ &= (t_{comp} + t_{comm}) * (m + p - 1) \\ t_{average} &= t_{total} / m \end{aligned}$$

لا يمكن التحليل كما كنا نفعل سابقاً من خلال تحديد عدد الحسابات المتوازية أو عدد الاتصالات المتوازية.

□ حالة نسخة واحدة من المسألة:

$$\begin{aligned} t_{comp} &= 1 \text{ (عملية حسابية واحدة في كل حلقة)} \\ t_{comm} &= 2(t_{startup} + t_{data}) \\ t_{total} &= (2(t_{startup} + t_{data}) + 1) p \end{aligned}$$

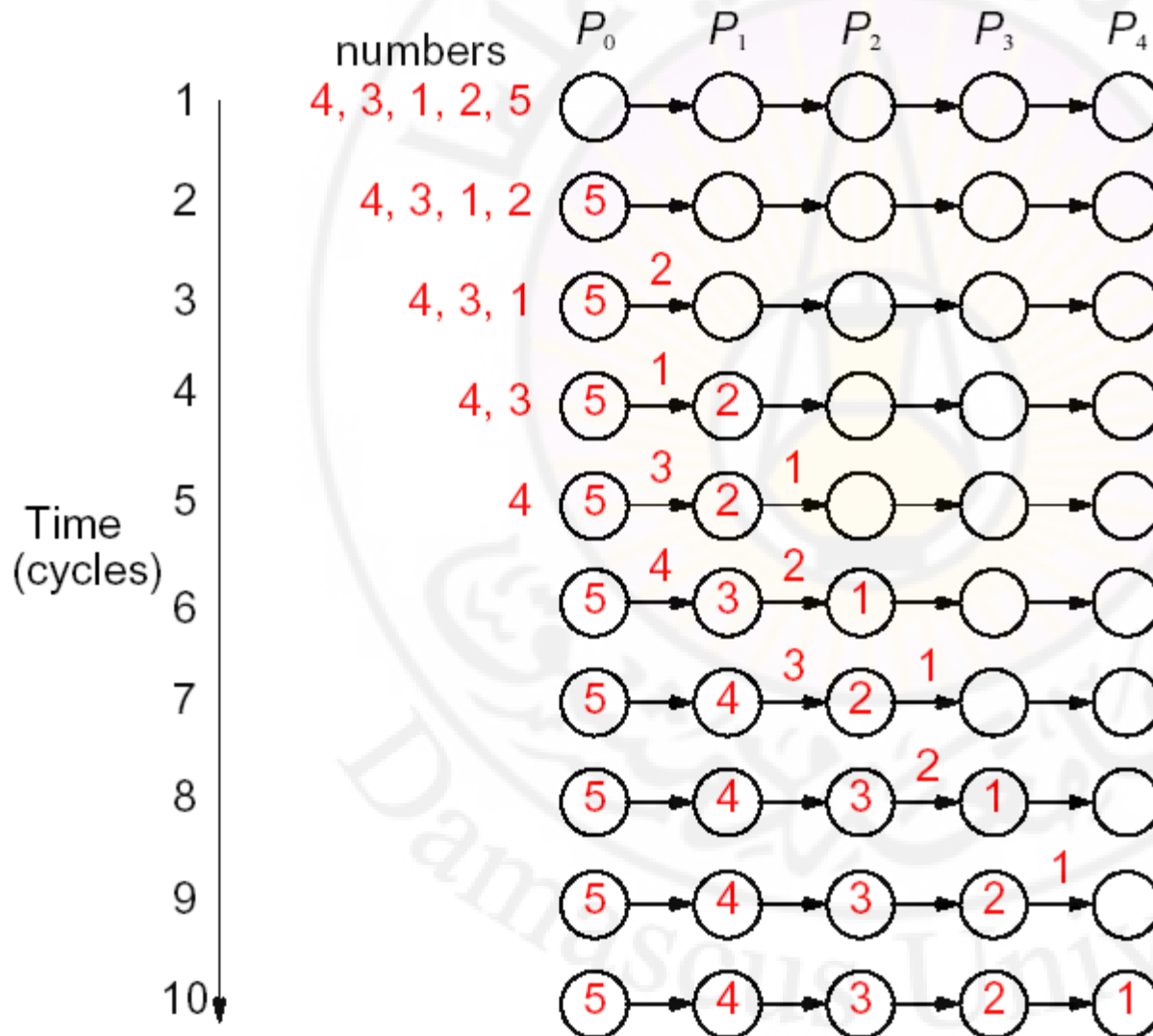
□ حالة عدة نسخ من المسألة:

$$\begin{aligned} t_{total} &= (2(t_{startup} + t_{data}) + 1) (m+n-1) \\ t_{average} &= t_{total} / m = 2(t_{startup} + t_{data}) + 1 \end{aligned}$$

يكافئ حلقة pipeline واحدة.

مثال (٢): فرز الأعداد

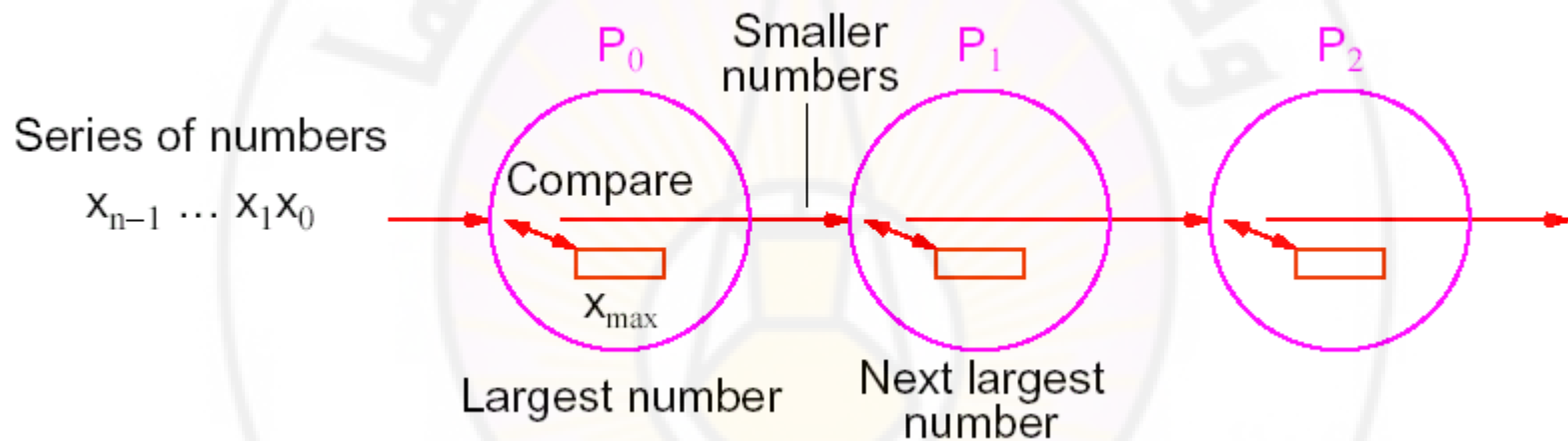
Example (2): Sorting Numbers



مثال تفرعي عن طريقة
الفرز بالإقحام
.Insertion Sort

مثال (٢): فرز الأعداد

Example (2): Sorting Numbers



الحساب باستخدام Pipeline من النوع الثاني.

مثال (٢): فرز الأعداد

Example (2): Sorting Numbers

خوارزمية الإجراء P_i :

```
recv( $P_{i-1}$ , number);  
  IF (number > x) {  
    send( $P_{i+1}$ , x);  
    x = number;  
  } ELSE send( $P_{i+1}$ , number);
```

إذا كان لدينا n عدد، فإن الإجراء i سيصله $n-i$ عدد وسوف يرسل $n-i-1$ عدد إلى الإجراءات التي تليه.

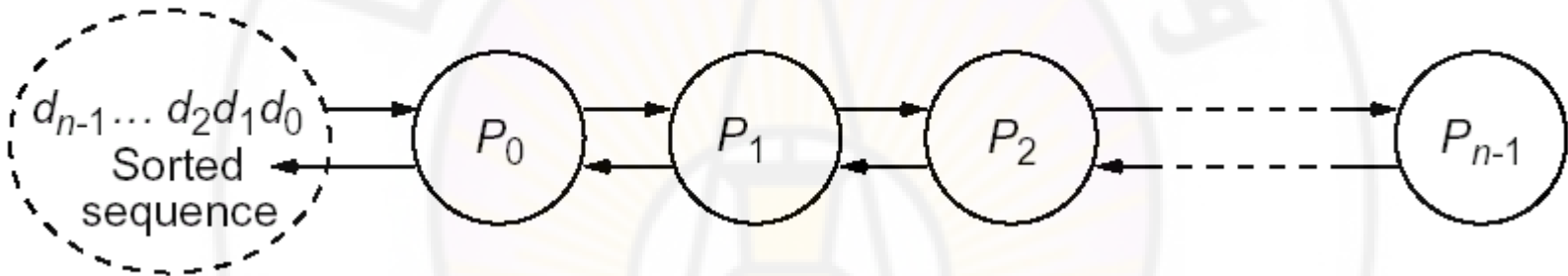
مثال (٢): فرز الأعداد

Example (2): Sorting Numbers

```
Right_procno = n - i - 1  
recv(Pi-1, x);  
for (j = 0; j < Right_procno; j++) {  
    recv(Pi-1, number);  
    IF (number > x) {  
        send(Pi+1, x);  
        x = number;  
    } ELSE send(Pi+1, number);  
}
```


مثال (٢): فرز الأعداد – إعادة النتائج Example (2): Sorting Numbers

Master process



تتميز هذه الطريقة بقدرتها على إرسال النتائج من قبل أي إجراء فور تلقيه
لآخر عدد.

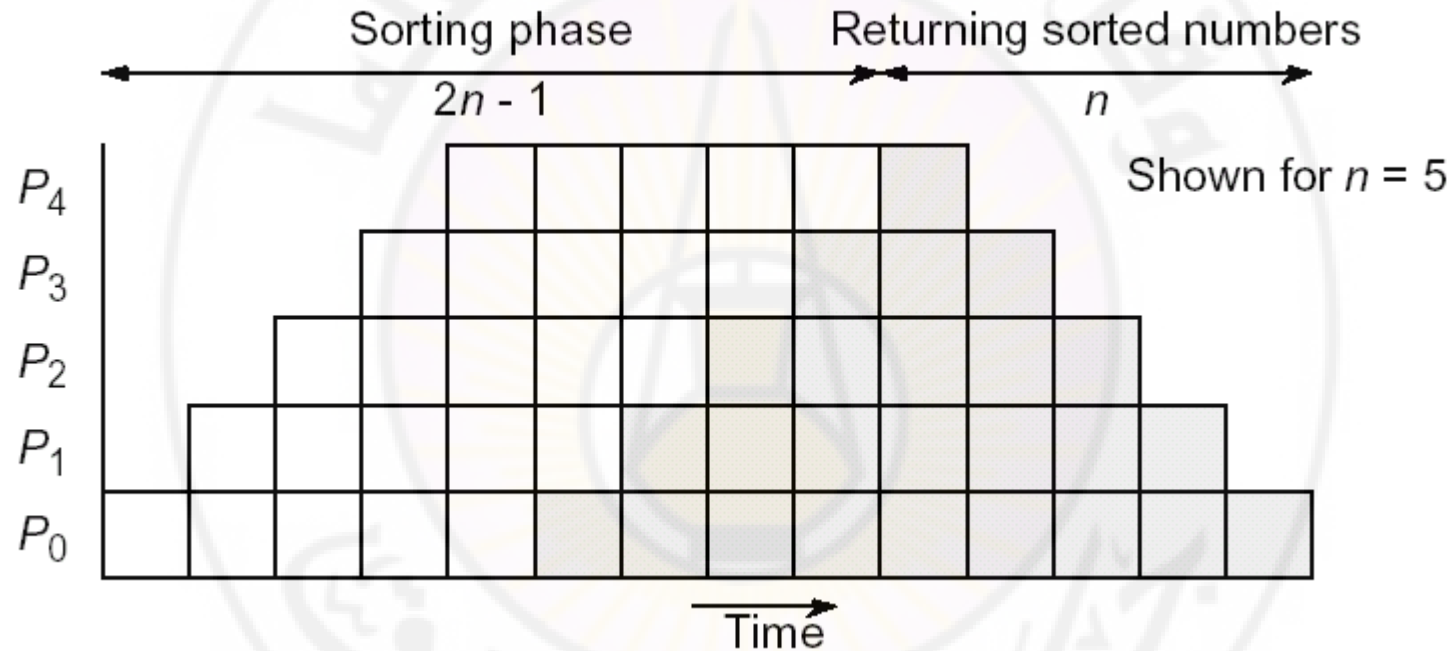
يمكن أيضاً استخراج النتائج باستخدام بنية الحلقة Ring
configuration.

مثال (٢): فرز الأعداد – إعادة النتائج

Example (2): Sorting Numbers

```
Right_procno = n - i - 1
recv(Pi-1, x);
for (j = 0; j < Right_procno; j++) {
    recv(Pi-1, number);
    IF (number > x) {
        send(Pi+1, x);
        x = number;
    } ELSE send(Pi+1, number);
}
Send(Pi-1, x);
for (j = 0; j < Right_procno; j++) {
    recv(Pi+1, x);
    send(Pi-1, x);
}
```

مثال (٢): فرز الأعداد – إعادة النتائج Example (2): Sorting Numbers



$t_{comp} = 1$ (عملية حسابية واحدة في كل حلقة)

$t_{comm} = 2(t_{startup} + t_{data})$

$t_{total} = (2(t_{startup} + t_{data}) + 1)(2n - 1)$ (دون إعادة النتائج)

مثال (٣): حل جملة من المعادلات الخطية

Example (3): Solving a System of Linear Equations

$$\begin{array}{rcl} a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 & \dots & + a_{n-1,n-1}x_{n-1} & = b_{n-1} \\ & & \cdot & \\ & & \cdot & \\ a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 & & & = b_2 \\ a_{1,0}x_0 + a_{1,1}x_1 & & & = b_1 \\ a_{0,0}x_0 & & & = b_0 \end{array}$$

الحساب باستخدام Pipeline من النوع الثالث.

مثال (٣): حل جملة من المعادلات الخطية

Example (3): Solving a System of Linear Equations

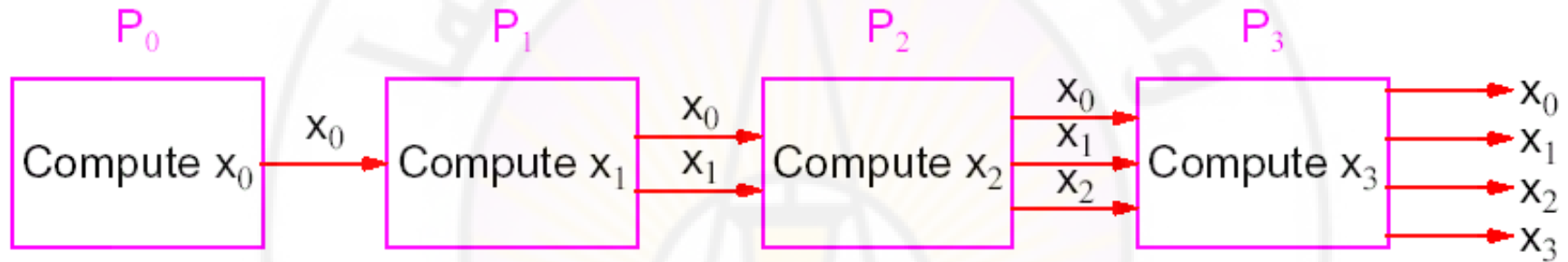
يمكن حل مجموعة المعادلات الخطية السابقة بأسلوب التعويض الخلفي على الشكل التالي:

$$x_0 = \frac{b_0}{a_{0,0}} \quad x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

مثال (٣): حل جملة من المعادلات الخطية

Example (3): Solving a System of Linear Equations



الحساب باستخدام Pipeline من النوع الثالث وفقاً للمعادلة التالية:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}$$

مثال (٣): حل جملة من المعادلات الخطية – الخوارزمية التسلسلية

Example (3): Solving a System of Linear Equations

```
x[0] = b[0] / a[0][0];  
for (i = 1; i < n; i++) {  
    sum = 0;  
    for (j = 1; j < i; j++) {  
        sum += a[i][j]*x[j];  
    }  
    x[i] = (b[i] - sum)/a[i][i];  
}
```

مثال (٣): حل جملة من المعادلات الخطية – الخوارزمية التفرعية

Example (3): Solving a System of Linear Equations

```
for (j = 1; j < i; j++) {  
    recv(Pi-1, x[j]);  
    send(Pi+1, x[j]);  
}  
sum = 0;  
for (j = 1; j < i; j++)  
    sum += a[i][j]*x[j];  
x[j] = (b[i] - sum)/a[i][j];  
send(Pi+1, x[j]);
```


مثال (٣): حل جملة من المعادلات الخطية – تحليل الخوارزمية

Example (3): Solving a System of Linear Equations

لا تتشابه المسائل في هذا النوع من ال Pipeline من حيث حجم الحسابات والاتصالات.

يقوم الإجراء P_0 بتنفيذ:

عملية تقسيم واحدة + عملية اتصال واحدة.

تقوم الإجراءات ($0 < i < n-1$) بتنفيذ:

($2i+1$) عملية اتصال + ($2i+2$) عملية حساب.

يقوم الإجراء الأخير P_{n-1} بتنفيذ:

($n+1$) عملية اتصال + ($2n-1$) عملية حساب.

زمن التنفيذ التفرعي = $O(n)$.

زمن التنفيذ التسلسلي = $O(n^2)$.

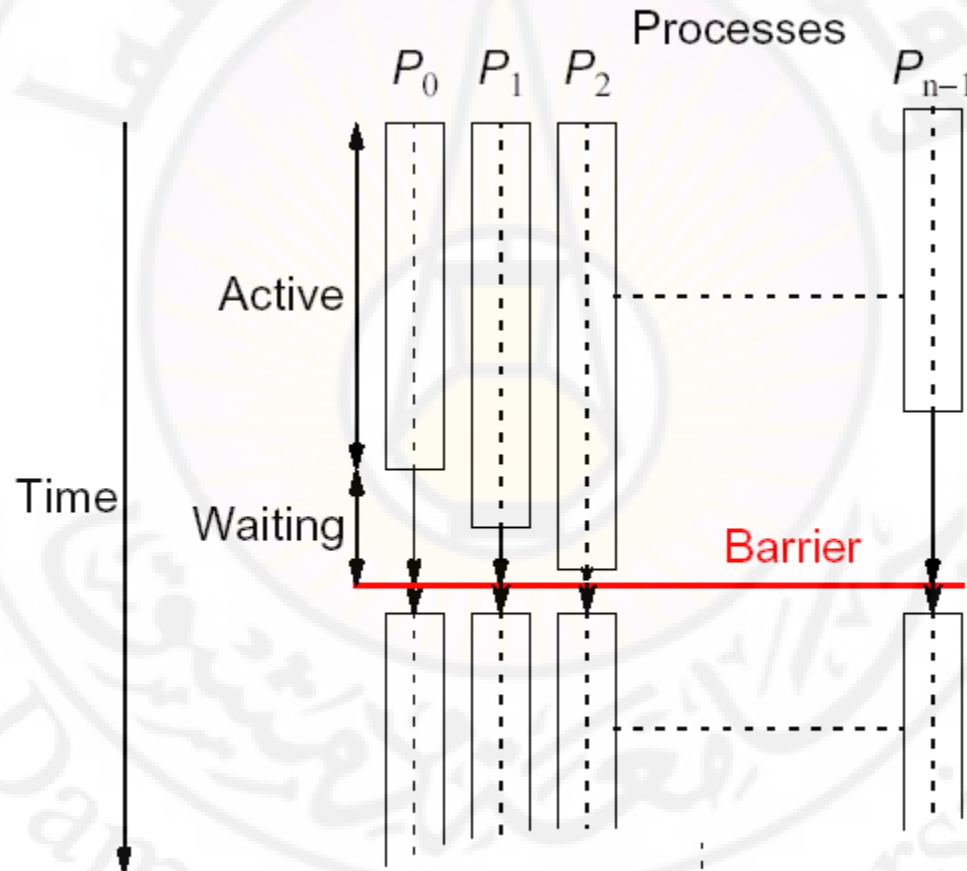
الحسابات المتزامنة

Synchronous Computations

- تتطلب التطبيقات التي تعتمد على المزامنة بين كافة الإجراءات المرتبطة بها، إجراء مزامنة بين هذه الإجراءات عند لحظات معينة Barriers.
- **حد التزامن Barrier** هو الطريقة المعتمدة لتحقيق التزامن، ويجري وضع هذا الحد عند النقطة التي يجب أن يصل إليها الإجراء المعني بعملية التزامن.
- يمكن لجميع الإجراءات أن تتابع تنفيذها بعد أن تصل كافة (أو بعض) الإجراءات إلى حد التزامن.

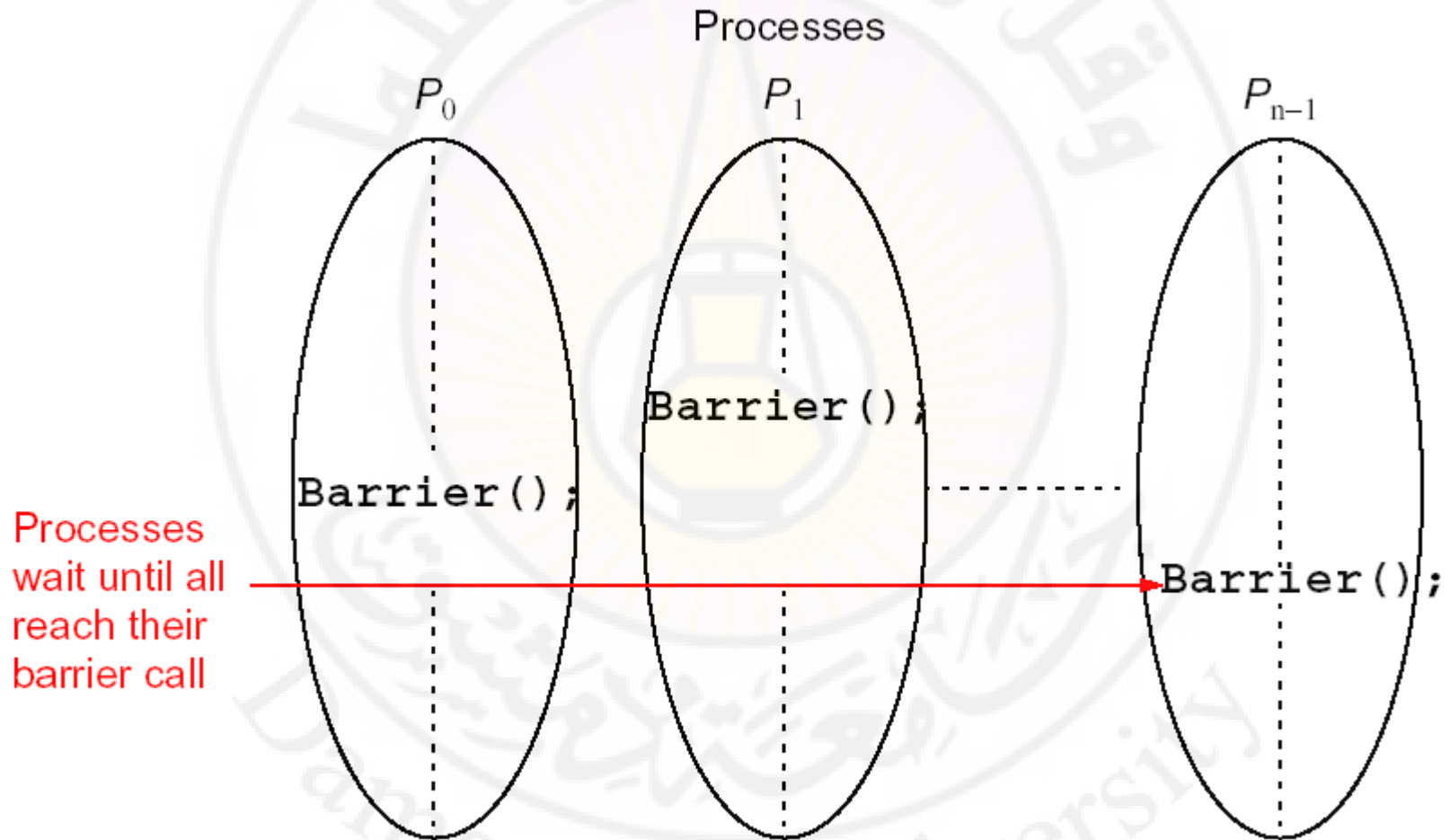
وصول الإجراءات إلى حد التزامن في لحظات مختلفة

Processes reaching barrier at different times



توابع تعريف التزامن

Barrier Routines



توابع تعريف التزامن

Barrier Routines

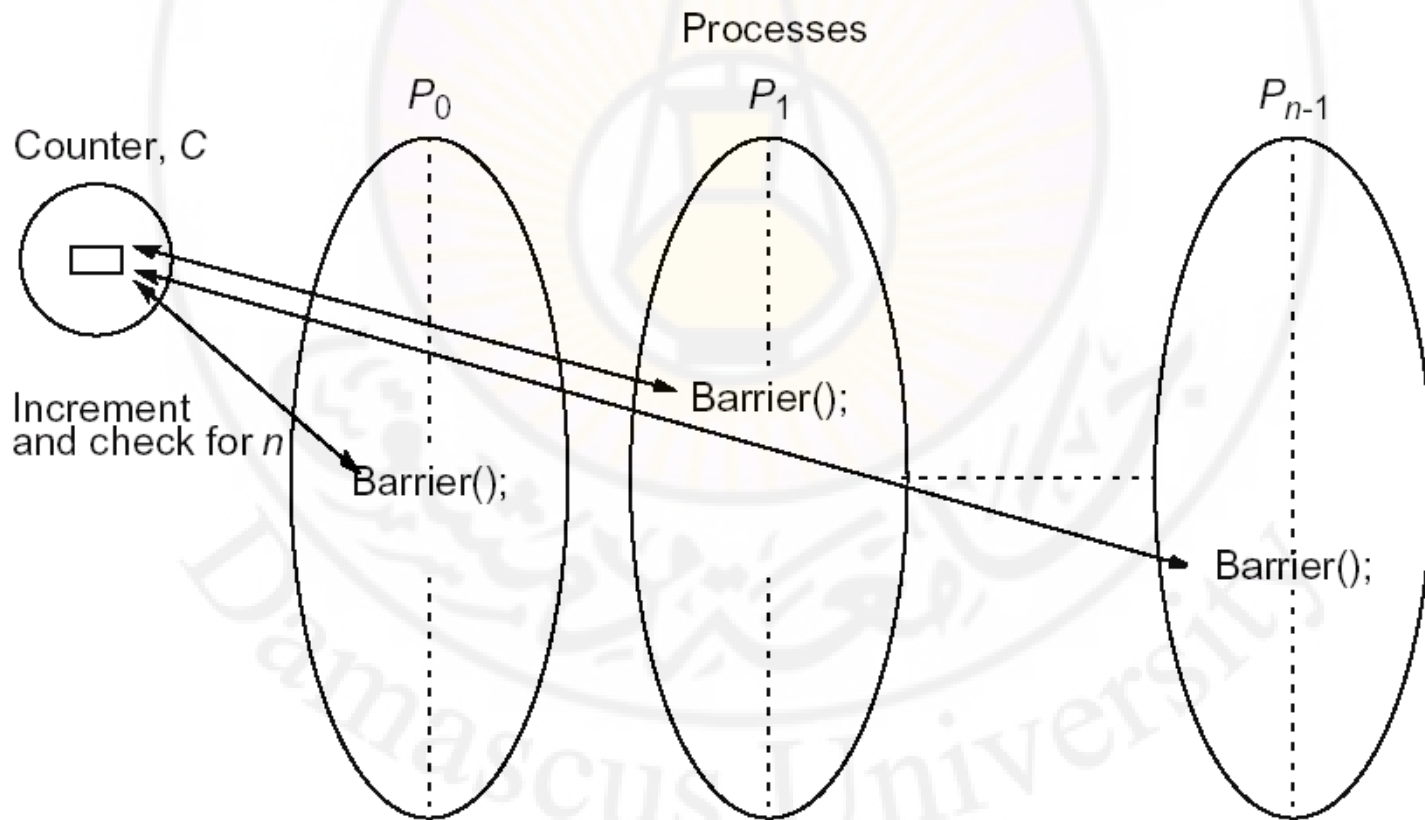
□ **MPI_Barrier()**: يأخذ هذا التابع وسيطاً واحداً وهو وسيط الاتصال communicator. يفيد التابع السابق في تحقيق عملية التزامن بين كافة الإجراءات التي تنتمي إلى مجموعة ما.

□ **PVM_Barrier()**: يشبه هذا التابع التابع السابق ولكن يمكن بواسطة هذا التابع أن نحدد عدد الإجراءات المشاركة بعملية التزامن.

تحقيق حد التزامن

Barrier Implementation

- حد التزامن الخطي Linear barrier ويمكن تحقيق ذلك بواسطة عداد مركزي.



تحقيق حد التزامن

Barrier Implementation

□ يمر حدود التزامن الخطية بمرحلتين، هما:

١. المرحلة الأولى – مرحلة الوصول: تصل الإجراءات إلى حد التزامن ولا تغادره حتى تصل كافة الإجراءات المعنية إليه.
٢. المرحلة الثانية – مرحلة المغادرة: تغادر الإجراءات بعدها حد التزامن.

ملاحظة: يجب أن يتمتع التحقيق الجيد لحد التزامن بالقدرة على استخدامه أكثر من مرة ضمن التطبيقات.

مثال برمجي

Example code

Master:

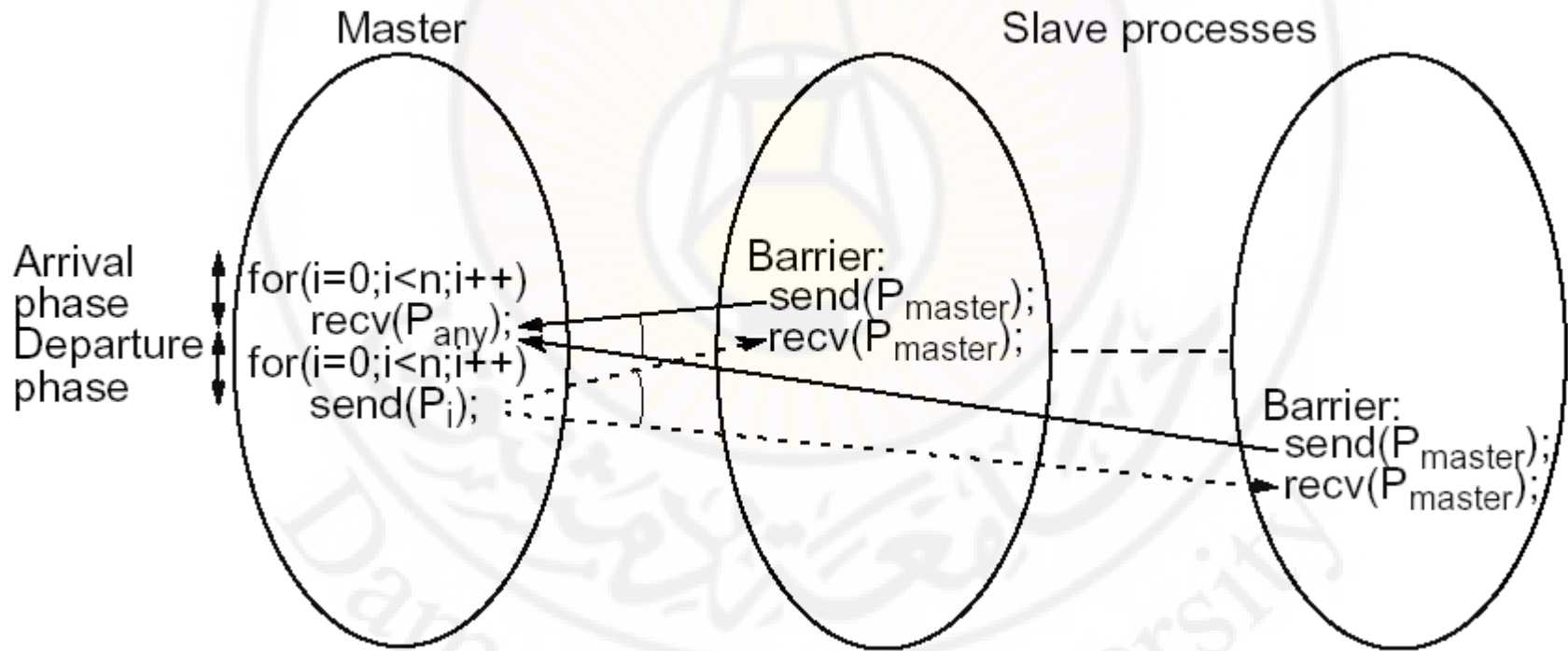
```
for (i = 0; i < n; i++) /*count slaves
    as they reach barrier*/
    recv(Pany) ;
for (i = 0; i < n; i++) /* release
    slaves */
    send(Pi) ;
```

Slave processes:

```
send(Pmaster) ;
recv(Pmaster) ;
```

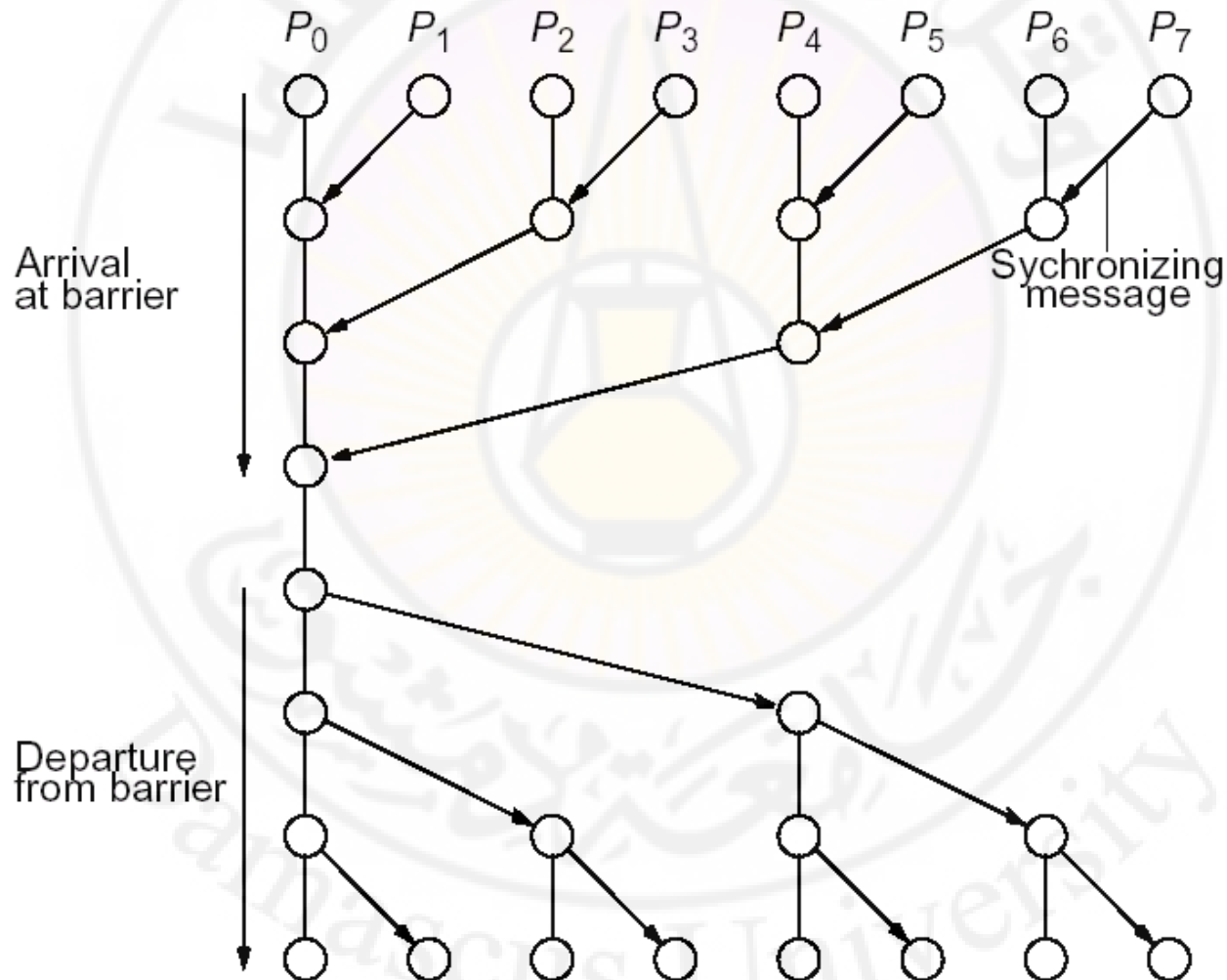

تحقيق حد التزامن ضمن نظام تبادل للرسائل

Barrier implementation in a message-passing system

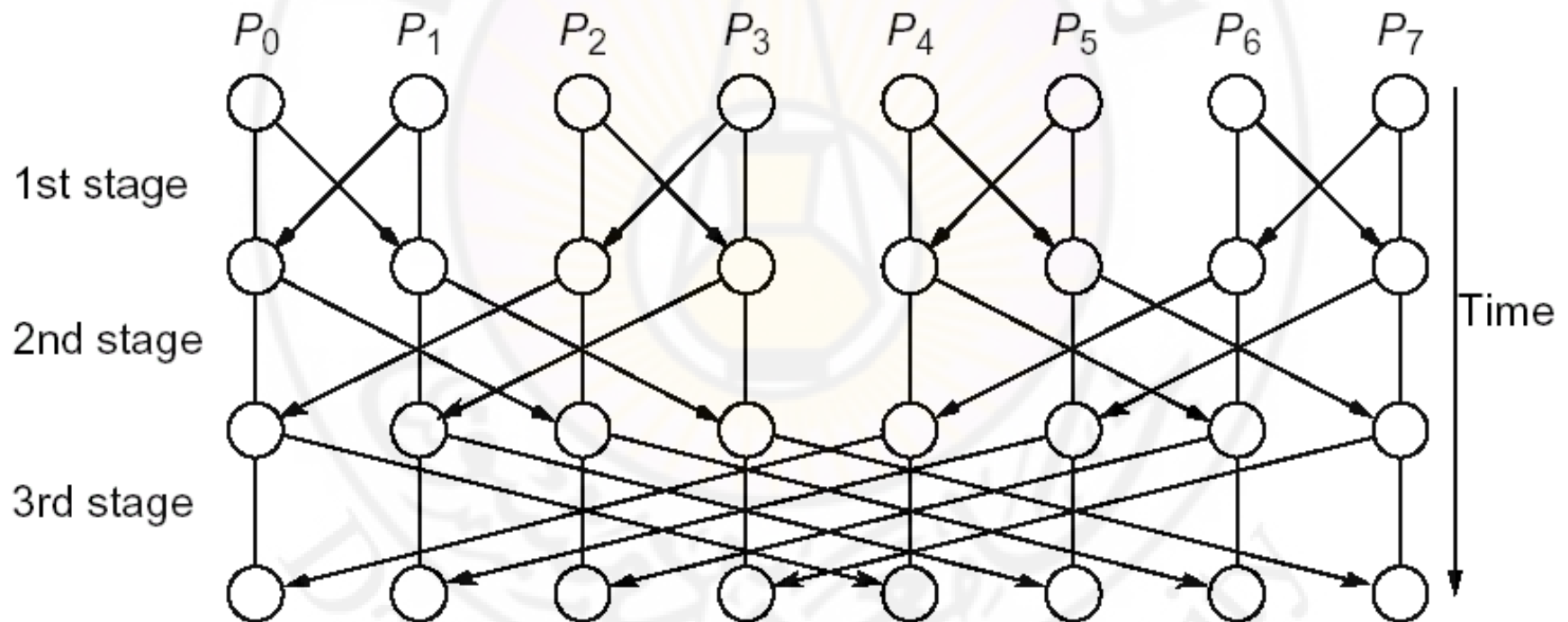


البنية الشجرية

Tree Implementation

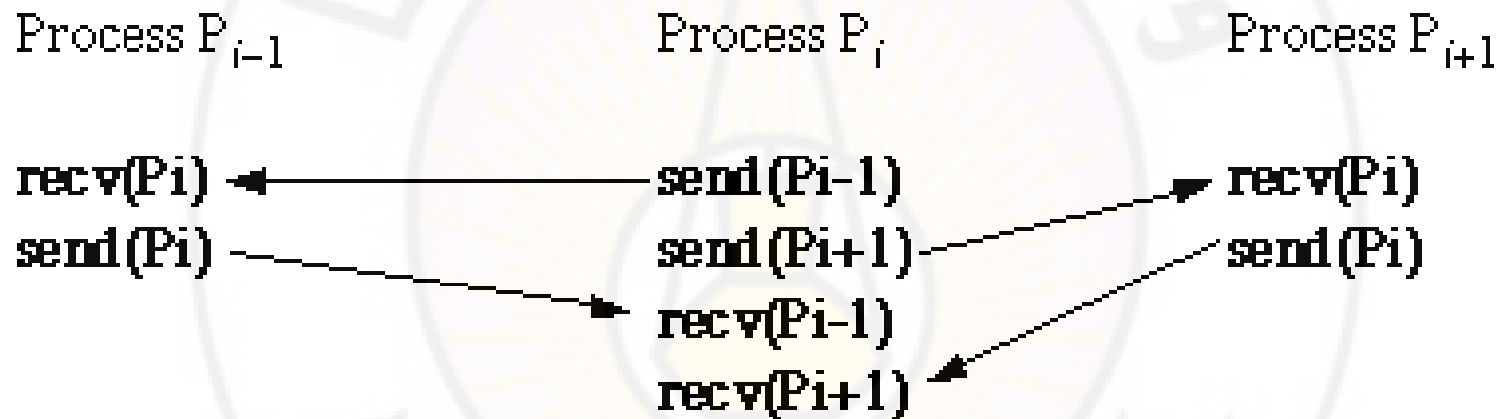


Butterfly Barrier



التزامن المحلي

Local Synchronization



لا يعتبر التزامن السابق أنه تزامن مثالي.
يمكن أن يحصل استعصاء Deadlock في بعض الأحيان (كيف يحدث ذلك وما هو الحل!).

التابع `sendrecv()` لتجاوز الاستعصاء

Process P_i

```
sendrecv (Pi) ;  $\longleftrightarrow$  sendrecv (Pi-1) ;  
                sendrecv (Pi+1) ;  $\longleftrightarrow$  sendrecv (Pi) ;
```

يتوفر ضمن المكتبة MPI التابعين: MPI_sendrecv() و MPI_sendrecv_replace() لتجاوز مشاكل الاستعصاء.

الحساب التفريعي على مستوى المعطيات

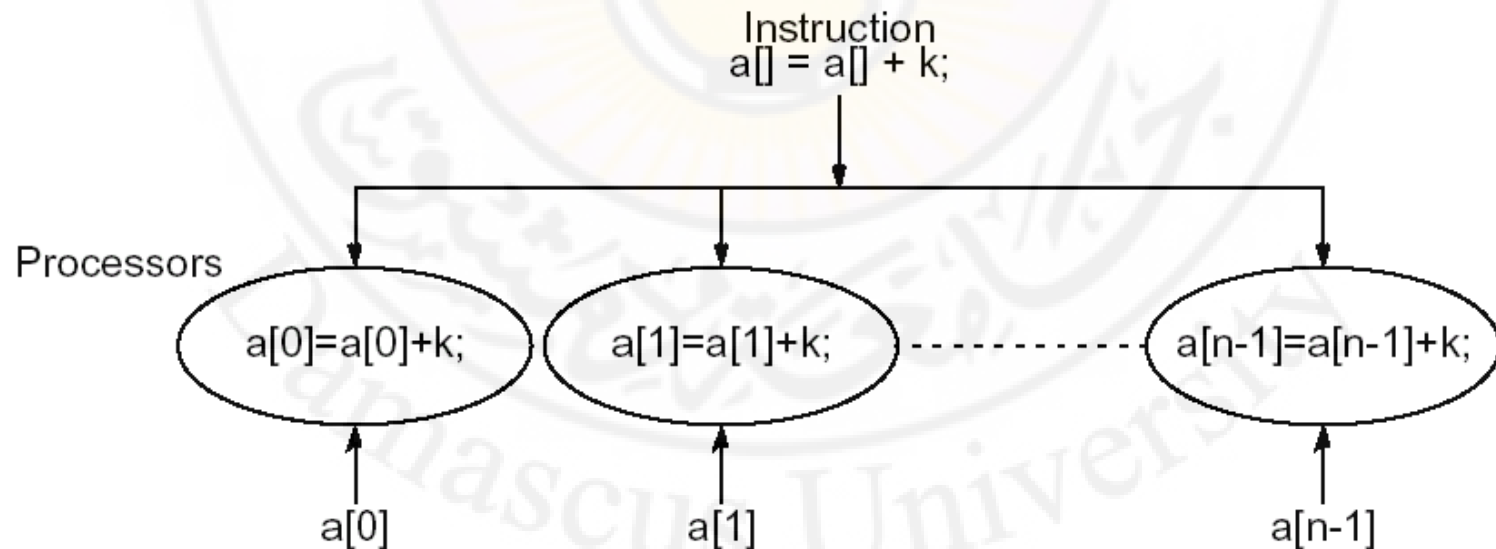
Data Parallel Computations

إضافة قيمة ثابتة إلى عناصر مصفوفة:

```
for (i=0; i<n; i++)
```

```
    a[i] = a[i] + k;
```

يمكن تنفيذ التعليمة الموجودة ضمن حلقة الـ for السابقة على التوازي وباستخدام عدة معالجات.



الحسابات المتزامنة

Synchronized Computations

يوجد لدينا نوعين من الحسابات المتزامنة، هما:

□ الحسابات ذات التزامن الشامل Fully synchronous حيث تشترك كافة الإجراءات بعملية التزامن.

□ مثال: تنفيذ نفس العملية مرتبطة بعناصر معطيات مختلفة على التوازي باستخدام عدة إجراءات.

□ يتصف هذا النوع من الحسابات بما يلي:

□ سهولة البرمجة

□ سهولة التوسع Scalability.

□ ملائم للعديد من التطبيقات.

□ الحسابات ذات التزامن المحلي Locally synchronous حيث تشمل عملية التزامن الإجراءات القريبة وليس كافة جميع الإجراءات.

البنية forall

forall construct

لدينا في بعض لغات البرمجة التفرعية البنية forall:

```
forall (i=0; i<n; i++) {  
    body  
}
```

تفيد هذه التعليمة أن هناك n نسخة من التعليمات ضمن جسم الحلقة والتي يمكن تنفيذها على التوازي بنفس الوقت. ويكون لدينا من أجل كل نسخة قيمة مختلفة للمتحول i (تأخذ النسخة الأولى القيمة $i=0$ وهكذا).

يمكن تطبيق تقنيات Data Parallel باستخدام متعددات الحواسيب أو المعالجات
مثال:

```
i = myrank;  
a[i] = a[i] + k; /* body */  
Barrier(mygroup);
```

حيث myrank هي رقم الإجراء ($0 \leq \text{myrank} \leq n-1$)

مثال للتفرع على مستوى المعطيات

Data Parallel Example

مسألة المجاميع الجزئية:
بفرض أنه لدينا قائمة من الأعداد

$$X_0, \dots, X_{n-1}$$

والمطلوب حساب كافة المجاميع الجزئية لها، أي:

$$X_0 + X_1, X_0 + X_1 + X_2, \dots$$

يلتئم هذا النوع من الحسابات العديد من المسائل، مثل: حسابات كثير الحدود،
الفرز وغيرها.

مسألة المجاميع الجزئية

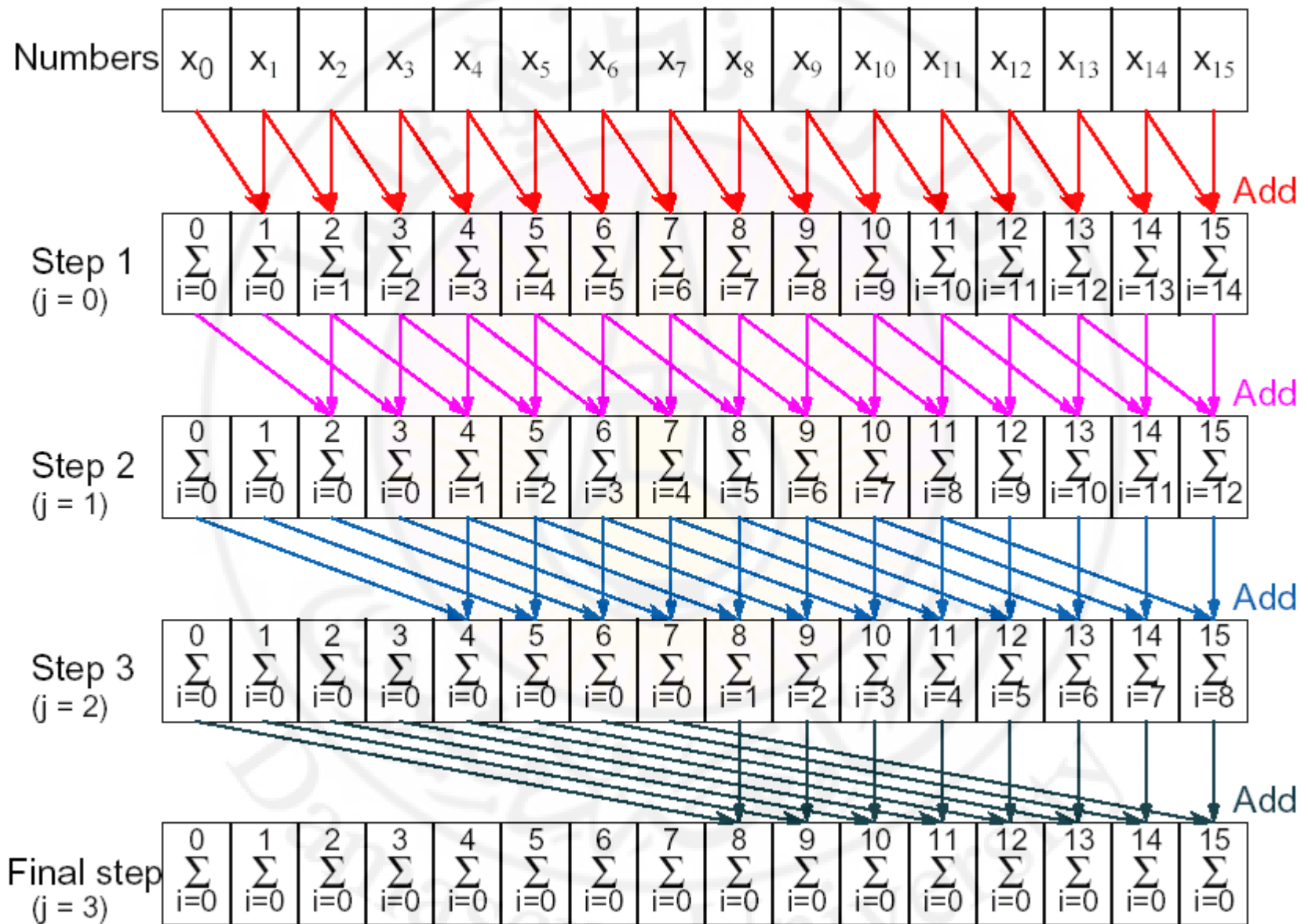
Partial Sums Problem

الخوارزمية التسلسلية

```
for (j=0; j<log(n); j++)  
    for (i=2j; i < n; i++)  
        x[i]=x[i]+x[i- 2j];
```

الخوارزمية التفرعية

```
for (j=0; j<log(n); j++)  
    forall (i=0; i < n; i++)  
        if (i ≥ 2j) x[i]=x[i]+x[i- 2j];
```



الحسابات المتزامنة التكرارية

Synchronous Iteration

يتألف كل تكرار من عدة إجراءات. تبدأ هذه الإجراءات عملها معاً في بداية التكرار ولا يبدأ التكرار التالي حتى تنتهي الإجراءات كافة من عملها المرتبط بالتكرار الحالي.

```
for (j=0; j< n; j++)          /* for each iteration          */
    forall (i=0; i < N; i++) {  /* N procs each using    */
        body(i);               /* specific value of l    */
    }
```

أو

```
for (j=0; j<n; j++) {
    i = myrank;
    body(i);
    Barrier(mygroup);
}
```

حل جملة من المعادلات الخطية

Solving a General System of Linear Equations

بفرض أنه لدينا جملة من المعادلات الخطية (n معادلة) من n مجهول والمطلوب إيجاد قيم هذه المجاهيل باستخدام الأسلوب التكراري.

$$\begin{array}{rcl} a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \dots & + a_{n-1,n-1}x_{n-1} & = b_{n-1} \\ \cdot & & \\ \cdot & & \\ \cdot & & \\ a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \dots & + a_{2,n-1}x_{n-1} & = b_2 \\ a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \dots & + a_{1,n-1}x_{n-1} & = b_1 \\ a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \dots & + a_{0,n-1}x_{n-1} & = b_0 \end{array}$$

حل جملة من المعادلات الخطية

Solving a General System of Linear Equations

يمكن استخدام الأسلوب التكراري من أجل حساب قيمة المجهول x_i على الشكل التالي:

$$x_i = \frac{1}{a_{i,i}} \left[b_i - \sum_{j \neq i} a_{i,j} x_j \right]$$

تعطي المعادلة السابقة قيمة المجهول x_i تبعاً لقيم المجاهيل الأخرى. يمكن بالتالي استخدام التكرار للوصول إلى تقريب مقبول لقيم المجاهيل.

حل جملة من المعادلات الخطية – الخوارزمية التسلسلية

Solving a General System of Linear Equations – Sequential Code

```
for (iteration = 0; iteration < limit; iteration++)  
    for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++) /* compute summation  
                                of a[i][j]x[j] */  
            if (i <> j) {  
                sum = a[i][j] * x[j];  
                x[i] = (b[i] - sum) / a[i][i];  
                /* Update unknown */  
            }
```

حل جملة من المعادلات الخطية – الخوارزمية التفرعية

Solving a General System of Linear Equations – Parallel Code

```
for (iteration = 0; iteration < limit; iteration++) {  
    for (j = 1; j < n; j++)  
        /* compute summation of a[i][j]x[j] */  
        if (i <> j) {  
            sum = a[i][j] * x[j];  
            x[i] = (b[i] - sum) / a[i][i]; /* Update unknown */  
        }  
    global_barrier();  
    bcast (x[i]);  
    gather(x);  
}
```

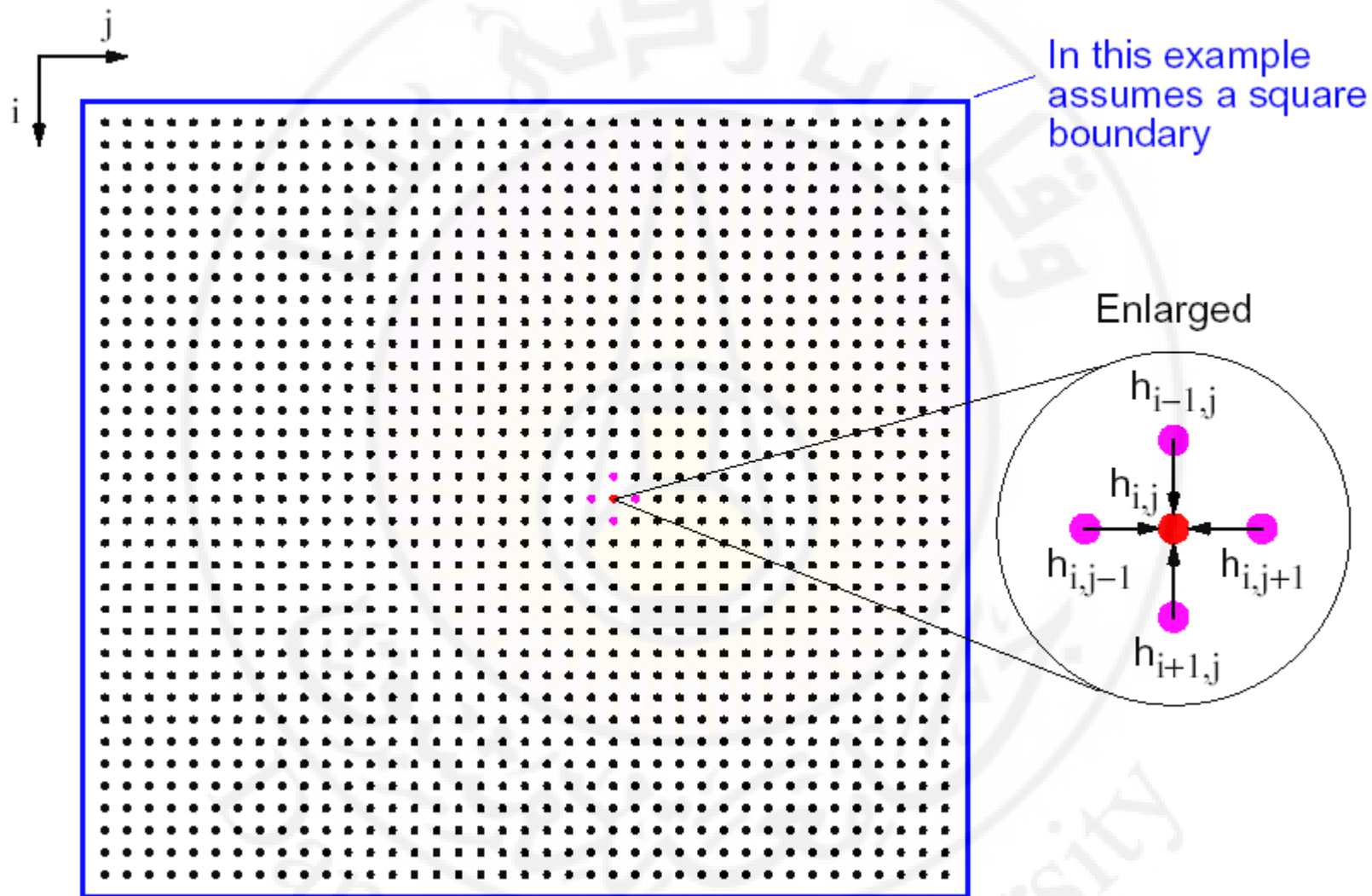

الحسابات ذات التزامن المحلي

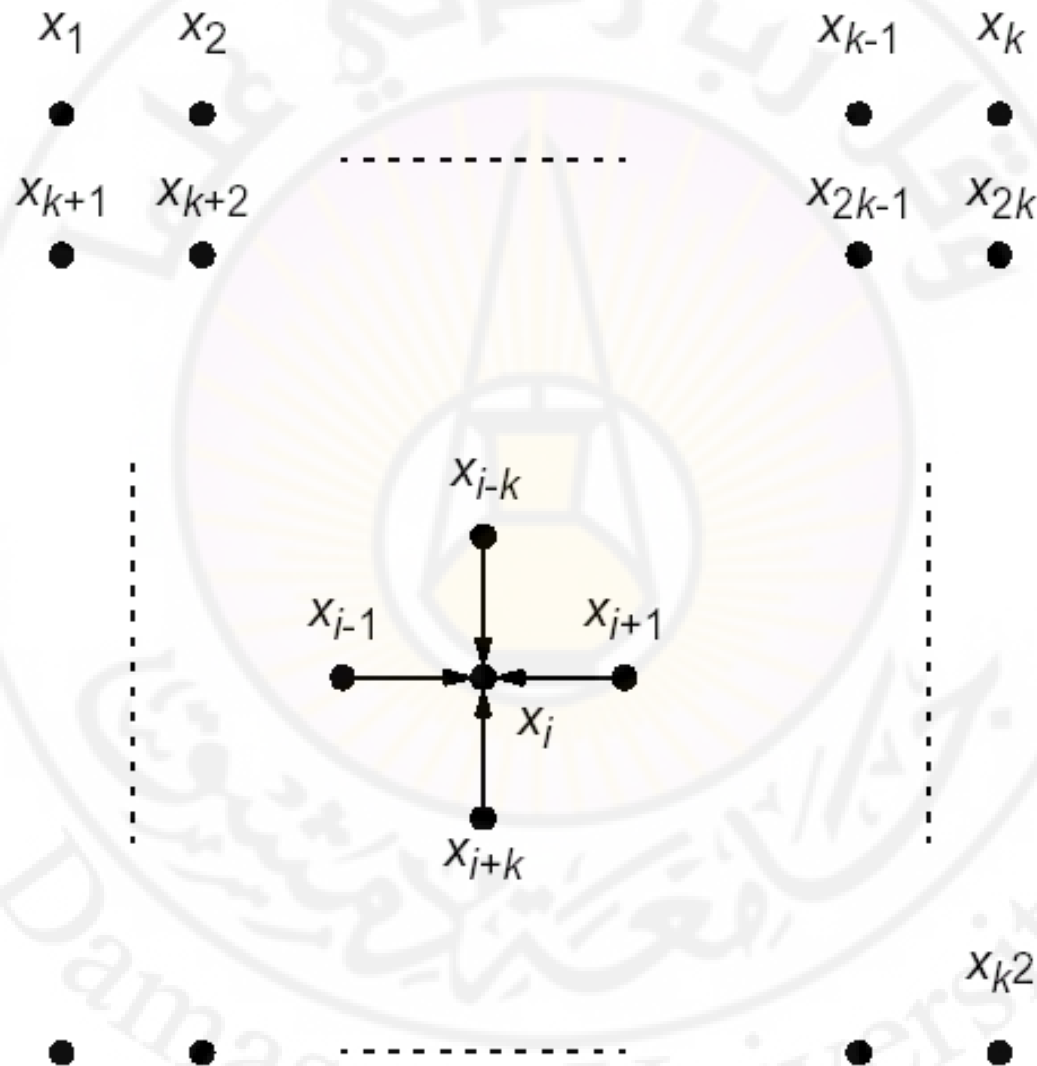
Locally Synchronous Computation

مسألة التوزيع الحراري:

يمكن حساب حرارة نقطة لصحيفة مربعة ذات حواف معلومة الحرارة باستخدام المعادلة التالية: $0 < i < n$ and $0 < j < n$

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$





الخوارزمية التسلسلية

Sequential Code

```
for (iteration = 0; iteration < limit; iteration++)  
    for (i = 1; i < n; i++)  
        for (j = 1; j < n; j++)  
            g[i][j] = (h[i-1][j] + h[i+1][j] + h[i][j-1] +  
                        h[i][j+1])/4;  
for (i = 1; i < n; i++)  
    /* update points */  
    for (j = 1; j < n; j++)  
        h[i][j] = g[i][j];
```

الخوارزمية التفرعية

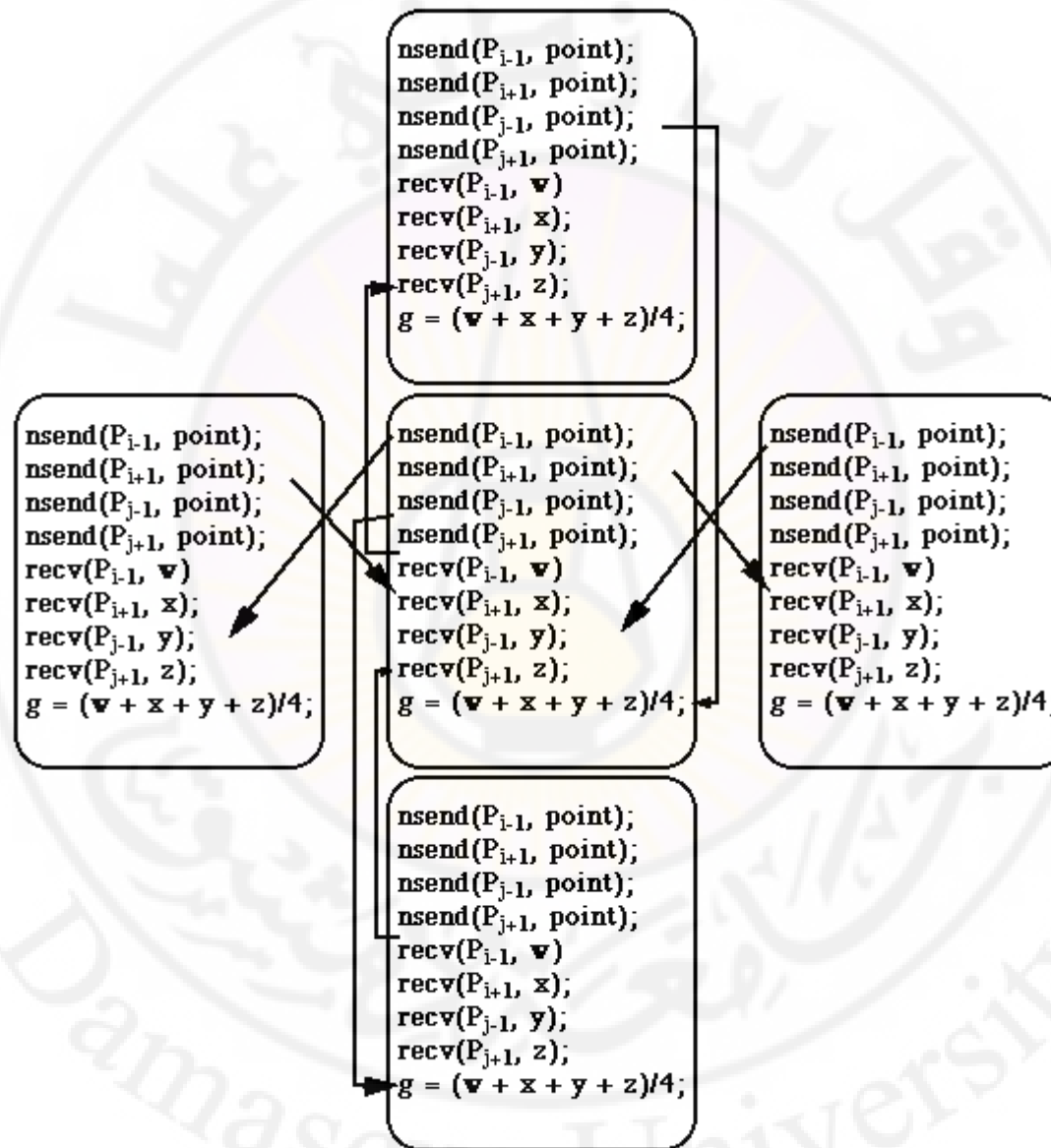
Parallel Code

```
do {  
    continue = FALSE;  
    for (i = 0; i < n; i++)  
        for (j = 0; j < n; j++)  
            if ((g[i][j] - h[i][j] > precision) ||  
                (h[i][j] - g[i][j] > precision)) {  
                g[i][j] = (h[i-1][j] + h[i+1][j] + h[i][j-1] +  
                    h[i][j+1])/4;  
                continue == TRUE; /* at least one point visited */  
            }  
    for (i = 1; i < n; i++) /* update points */  
        for (j = 1; j < n; j++)  
            h[i][j] = g[i][j];  
} while (continue == TRUE);
```

الخوارزمية التفرعية

Parallel Code

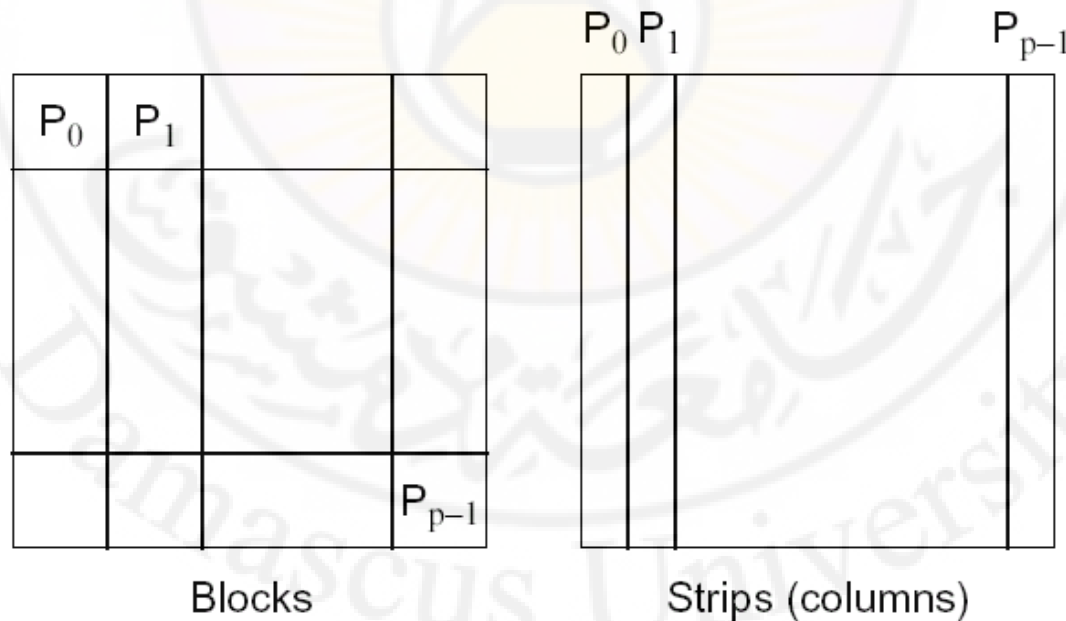
```
g = 0;
for (iteration = 0; iteration < limit; iteration++) {
    nsend(Pi-1, point); /* non-blocking sends */
    nsend(Pi+1, point);
    nsend(Pj-1, point);
    nsend(Pj+1, point);
    recv(Pi-1, w);      /* blocking receives */
    recv(Pi+1, x);
    recv(Pj-1, y);
    recv(Pj+1, z);
    g = (w + x + y + z)/4;
}
```



التجزئة Partitioning

مسألة التوزيع والتجزئة:

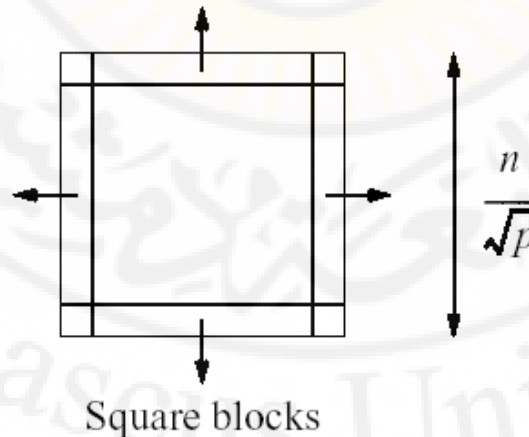
من الطبيعي أن يجري توزيع أكثر من نقطة على المعالجات. ويمكن أن تجري التجزئة على شكل كتل *blocks* أو أشرطة *strips*.



التجزئة على شكل كتل Block Partition

يجري تبادل المعلومات عبر الحواف الأربعة للكتلة.
تعتبر المعادلة التالية عن حجم عمليات الاتصال التي يمكن أن تجري بين كتلة والكتل المجاورة لها (تعتبر هذه المعادلة إذا كان لدينا $p \geq 9$):

$$t_{\text{commsq}} = 8 \left(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}} \right)$$



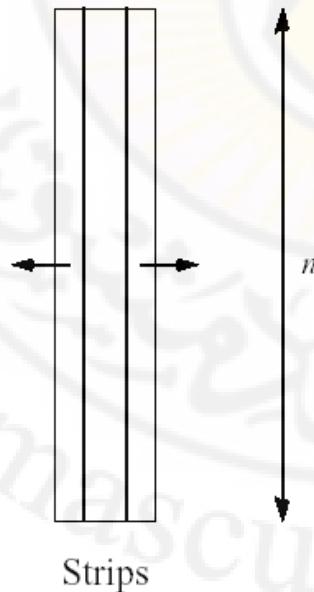
التجزئة على شكل أشرطة

Strip Partition

يجري تبادل المعلومات عبر حافظتين.

تعتبر المعادلة التالية عن حجم عمليات الاتصال التي يمكن أن تجري بين شريط وشريطين مجاورين له:

$$t_{\text{commcol}} = 4(t_{\text{startup}} + nt_{\text{data}})$$



الخيار الأمثل Optimum

يعتبر خيار التجزئة على شكل أشربة جيداً إذا كان الزمن $t_{startup}$ كبيراً. ويمكن في الحالة المعاكسة اللجوء إلى خيار التجزئة الكتلية. يمكن أن يكون حجم عمليات الاتصال في حالة التوزيع الكتلي أكبر إذا تحققت المعادلة التالية:

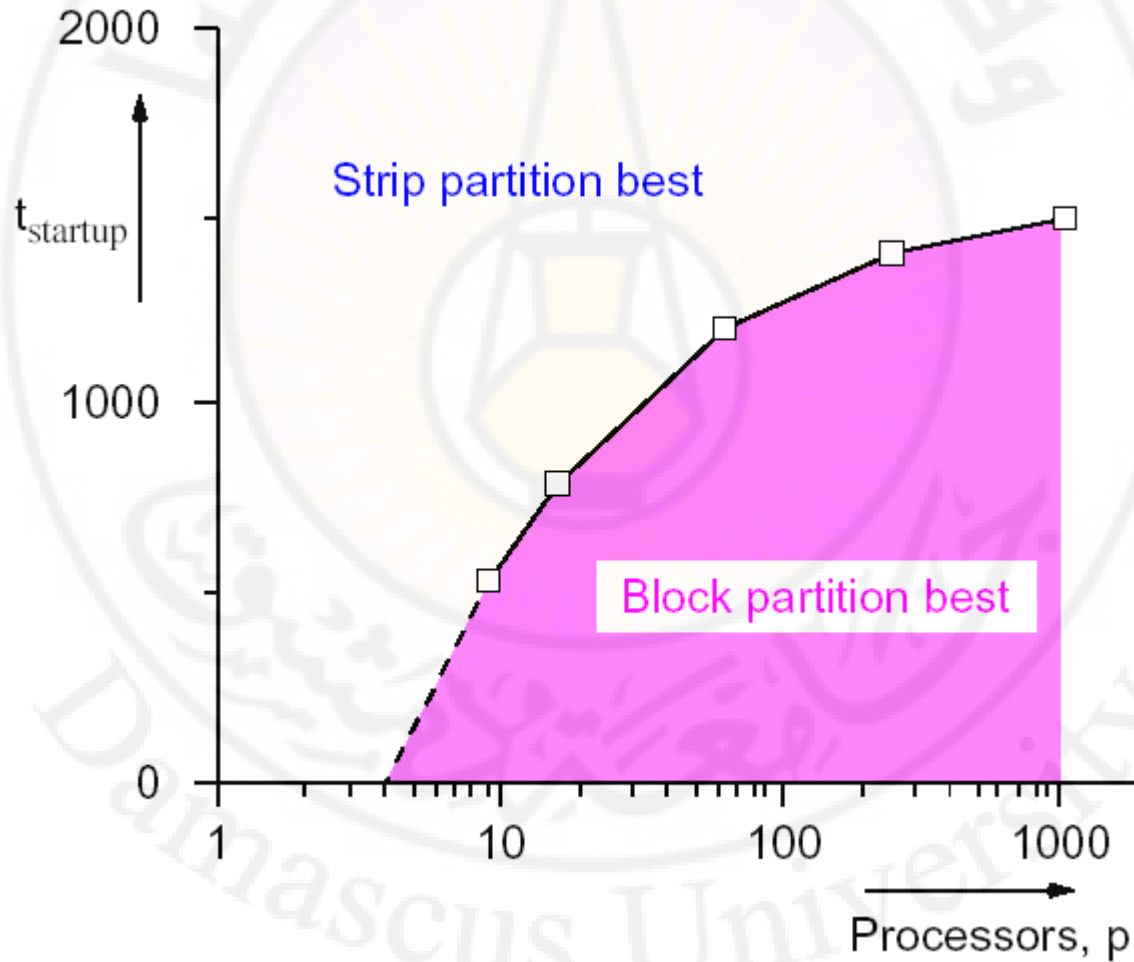
$$8\left(t_{startup} + \frac{n}{\sqrt{p}}t_{data}\right) > 4(t_{startup} + nt_{data})$$

or

$$t_{startup} > n\left(1 - \frac{2}{\sqrt{p}}\right)t_{data}$$

$(p \geq 9)$.

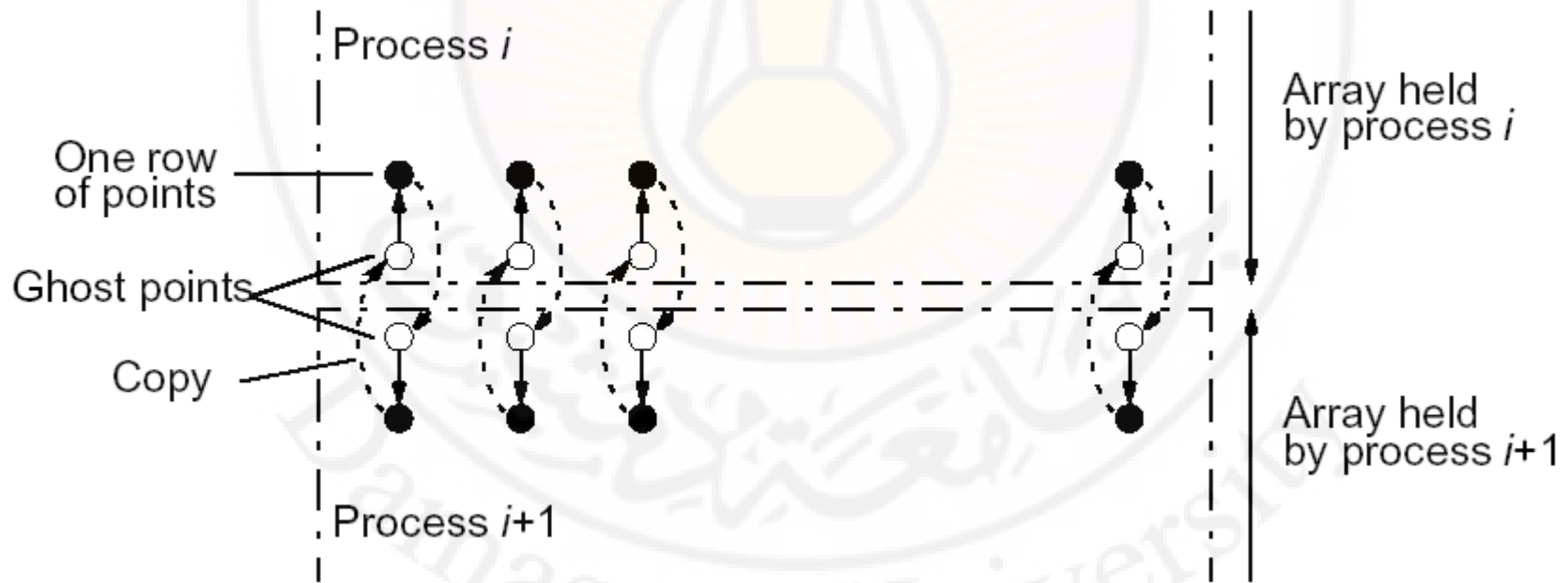
الخيار الأمثل Optimum



النقاط الوهمية / الشبح

Ghost Points

تساعد عملية إضافة سطر إضافي على مستوى كل حافة على تحسين الأداء وتجاوز العديد من الأشكال.

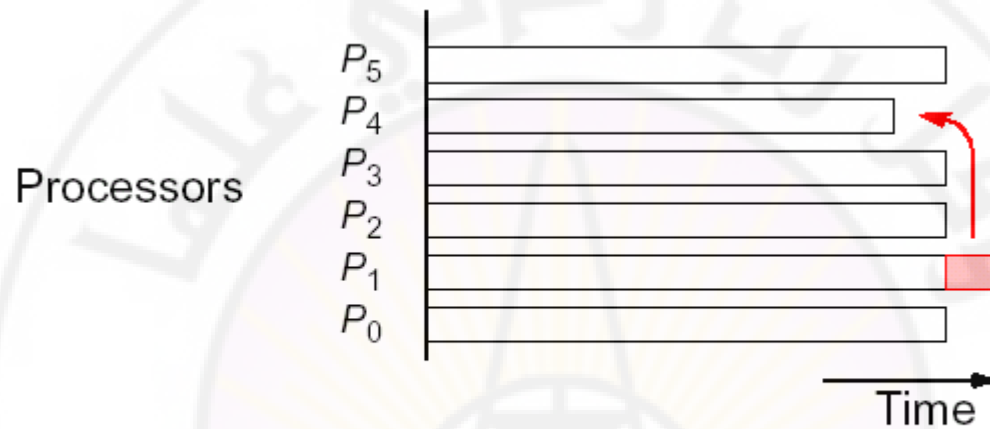


توزيع الحمل وتحديد لحظة إنتهاء التنفيذ

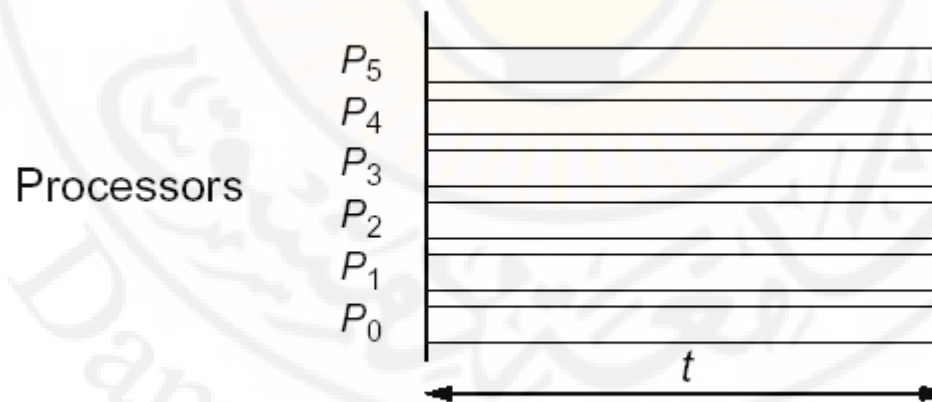
Load Balancing and Termination Detection

□ يستخدم **توزيع الحمل** من أجل توزيع الحسابات بشكل عادل على المعالجات بغية الحصول على أسرع زمن للتنفيذ. يمكن أن يجري توزيع الحمل بصورة مسبقة **Statically** أو بصورة متغيرة **Dynamically**.

□ **إنتهاء التنفيذ** هي اللحظة التي تنتهي فيها الحسابات لدى كافة المعالجات، وهي لحظة يصعب تحديدها عندما تكون عمليات الحساب موزعة.



(a) Imperfect load balancing leading to increased execution time



(b) Perfect load balancing

توزيع الحمل المسبق

Static Load Balancing

□ يجري هذا التوزيع قبل تنفيذ أي إجراء. ويستخدم لتنفيذ العديد من التقنيات المعروفة، مثل:

□ خوارزمية Round-Robin.

□ الخوارزميات العشوائية Randomized Algorithms.

□ التقطيع المتتالي Recursive Bisection.

□ محاكاة الإحماء Simulated Annealing.

□ الخوارزميات الجينية Genetic Algorithms.

□ تعاني هذه الطريقة من الإشكالات التالية:

□ من الصعب أن نقدر مسبقاً حجم الأعمال المنفذة باستخدام معالج ما.

□ يتغير حجم عمليات الاتصال تبعاً للظروف.

□ تتطلب بعض المسائل تنفيذ العديد من الخطوات الوسيطة للوصول إلى حلها.

توزيع الحمل الديناميكي

Dynamic Load Balancing

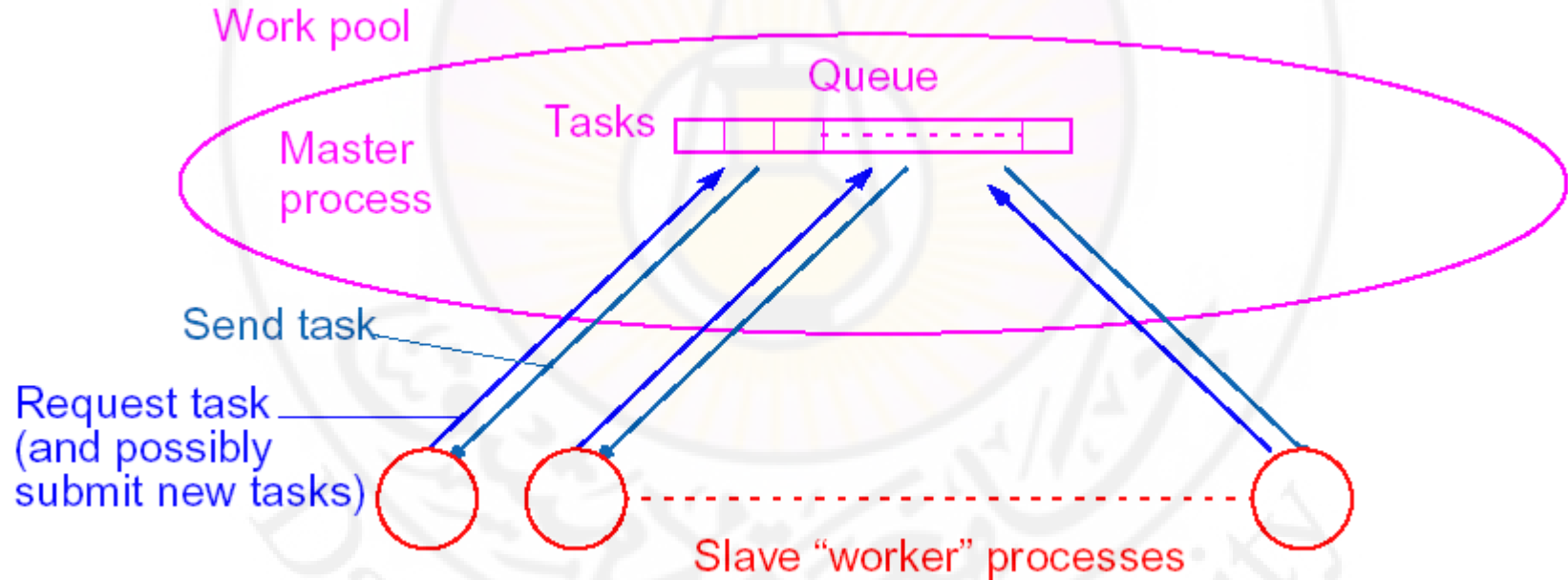
□ يجري إعادة توزيع الحمل ديناميكياً أثناء التنفيذ. تتم العملية بعد أن نأخذ بعين الاعتبار العديد من العوامل المرتبطة بظروف التنفيذ.

□ يؤدي هذا النوع التوزيع إلى ضياع بعض الوقت أثناء التنفيذ، لكنه يعتبر أكثر جدوى من التوزيع المسبق للحمل.

□ يمكن أن يتم بصورة مركزية Centralized أو لا مركزية Decentralized.

توزيع الحمل الديناميكي المركزي

Centralized Dynamic Load Balancing



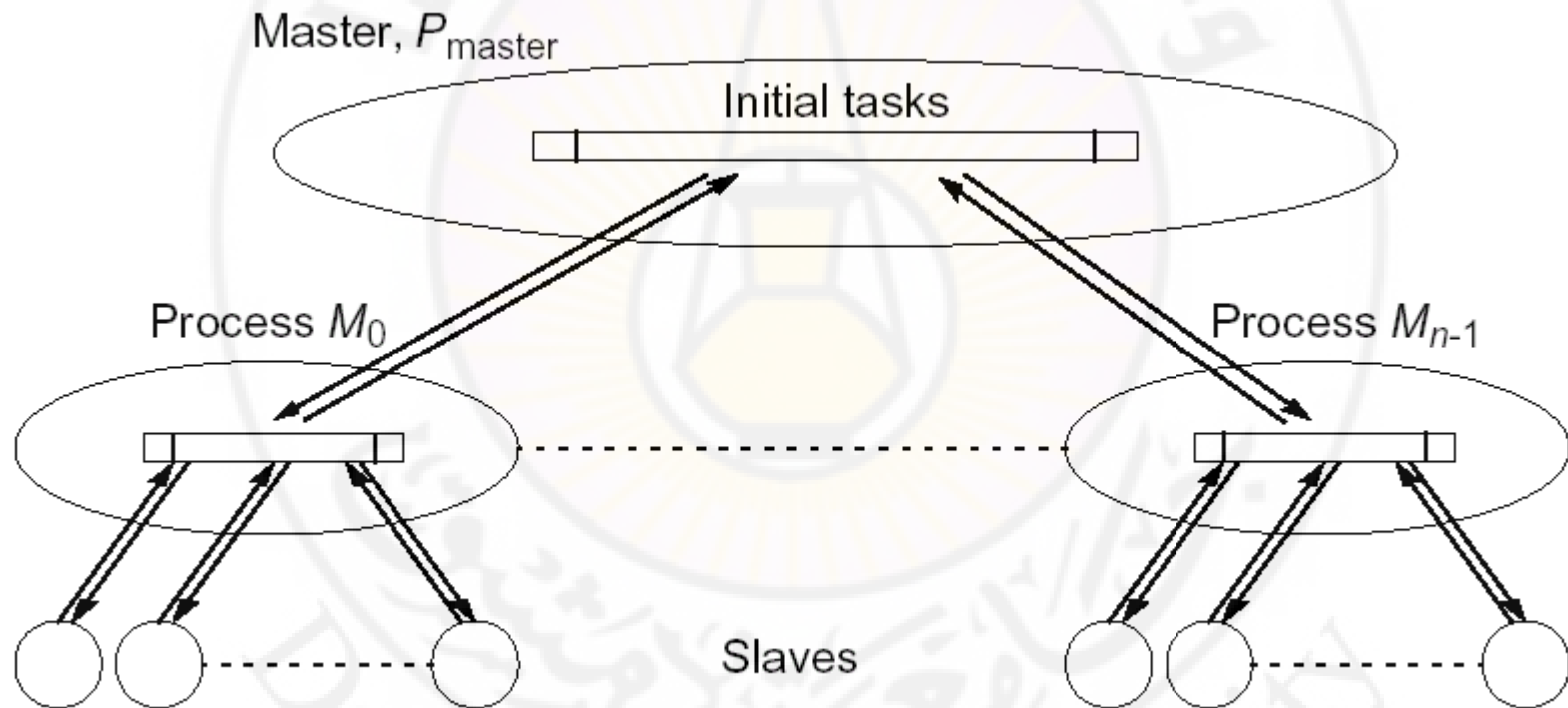
توزيع الحمل الديناميكي المركزي

Centralized Dynamic Load Balancing

- يتوفر لدى الإجراء الرئيسي مكان مركزي لتخزين المهام المراد تنفيذها Work Pool. غالباً ما يكون هذا الخزان عبارة عن صف إنتظار Queue (أي يجري توزيع مهمة واحدة في كل مرة).
- يجري إرسال العمل إلى الإجراءات الزبونة بعد أن تنتهي من المهام الموجودة لديها.
- يجري تحسس نهاية التنفيذ Termination:
 - عندما يكون مكان التخزين الرئيسي فارغاً
 - وعندما تقوم كافة الإجراءات الزبونة بطلب مهمة وليس لديها القدرة على توليد أية مهمة جديدة.

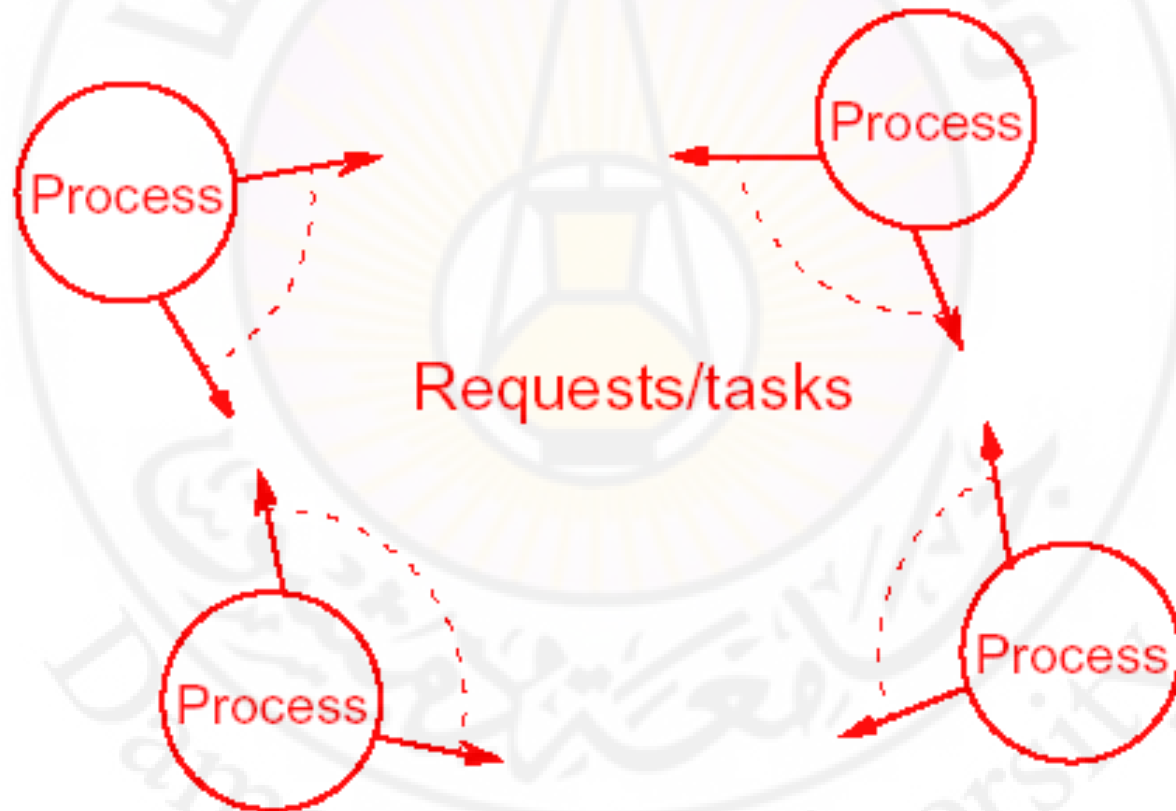
توزيع الحمل الديناميكي غير المركزي

Decentralized Dynamic Load Balancing



خزان المهام الموزع تماماً

Fully Distributed Work Pool



أساليب توزيع العمل / المهام

Task Transfer Mechanisms

يوجد لدينا طريقتين لتوزيع المهام:

□ Receiver-initiated Method حيث يقوم الإجراء بطلب مهام جديدة عند ملاحظته لوجود عدد قليل أو عدم وجود أي منها لديه. ثم يقوم باختيار الإجراء الذي سيأخذ منه مثل هذه المهام.

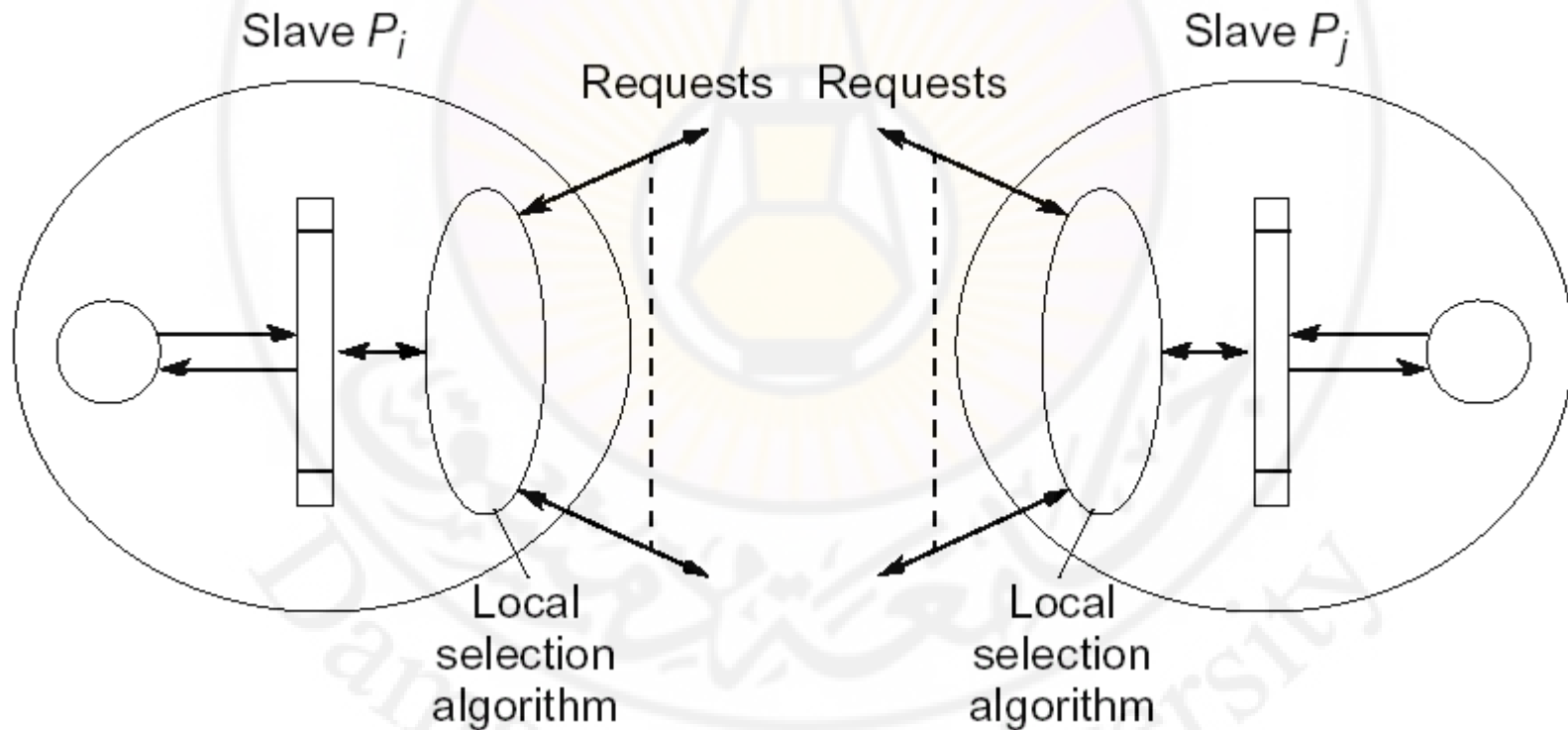
□ Sender-initiated Method: حيث يقوم الإجراء باختيار إجراء آخر ليرسل إليه جزء من عمله.

يمكن أن يتم اختيار الإجراءات وفقاً لعدة أساليب.

خوارزمية غير مركزية لاختيار الإجراء الزبون

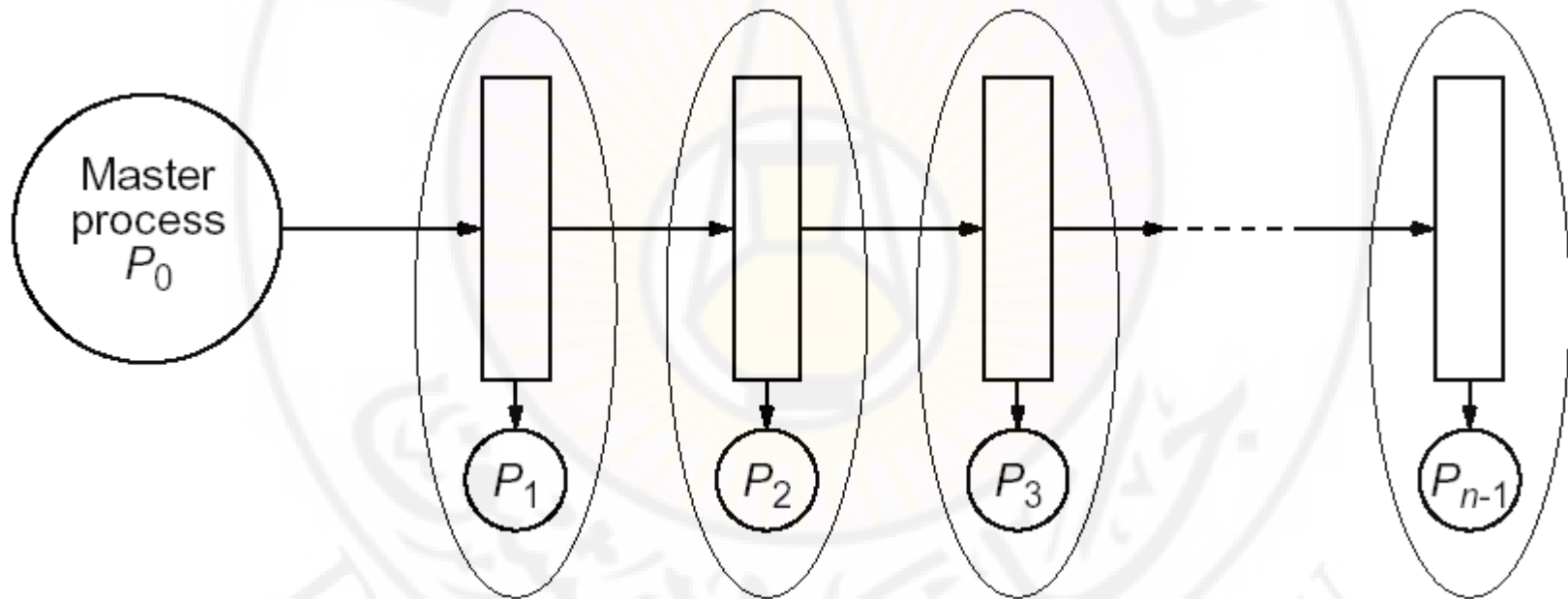
Decentralized Selection Algorithm

Requesting Tasks between Slaves



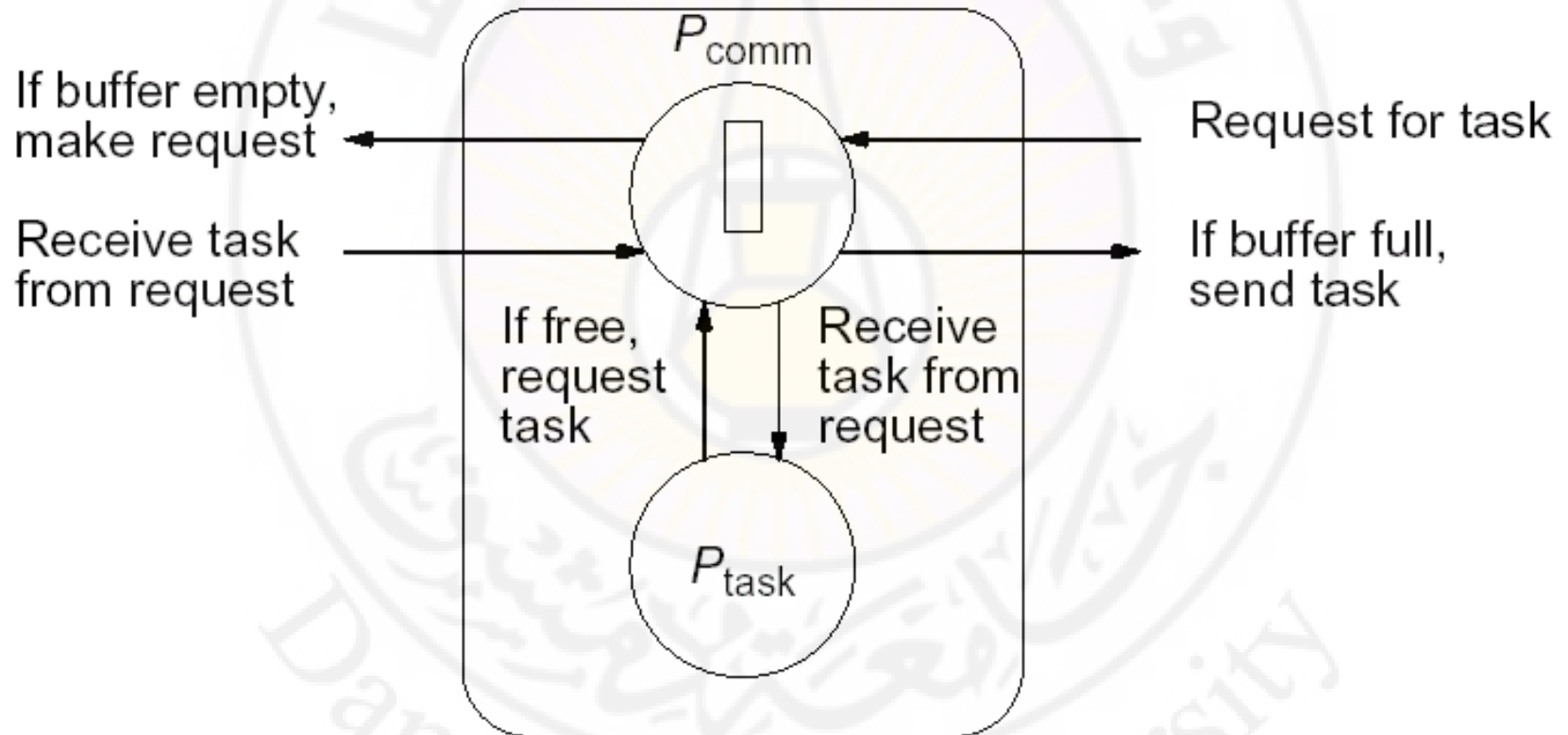
توزيع الحمل الديناميكي باستخدام بنية خطية

Load Balancing Using a Line Structure



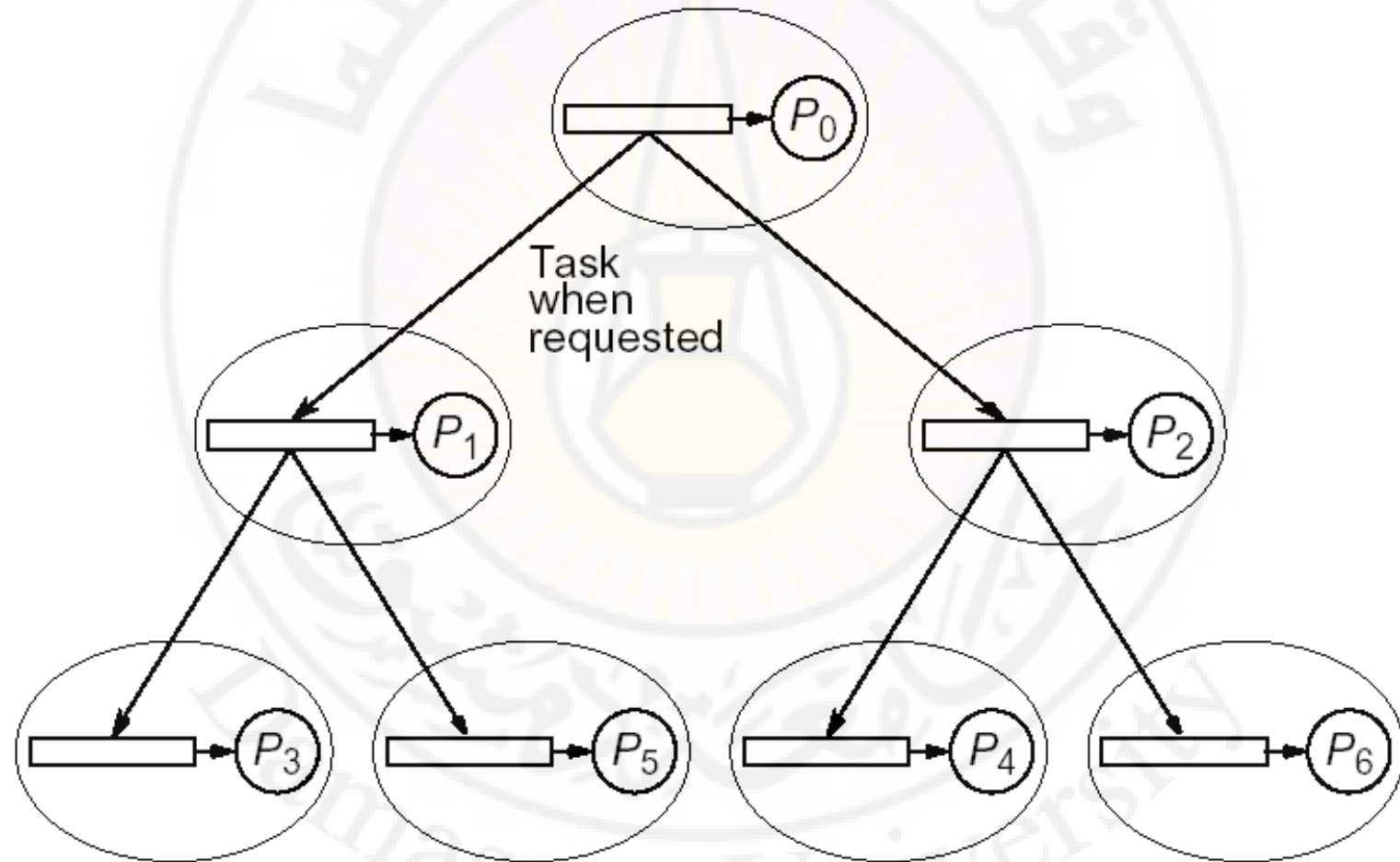
إزاحة الأفعال

Shifted Actions



توزيع الحمل الديناميكي باستخدام بنية شجرية

Load Balancing Using a Tree Structure



شروط إنهاء التنفيذ

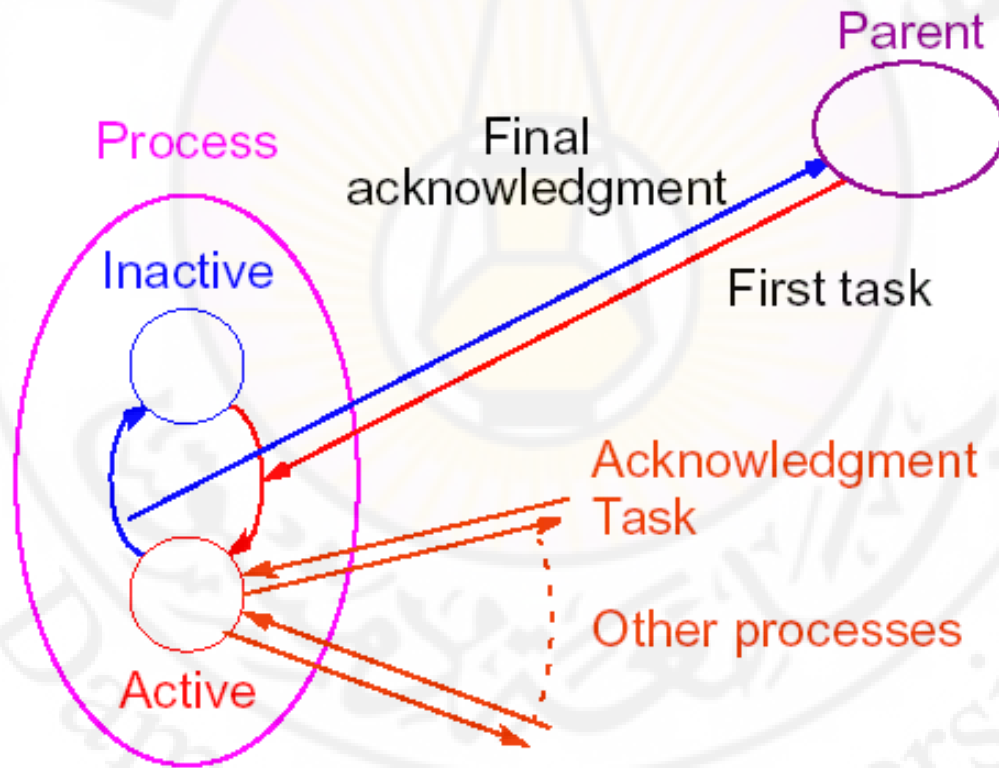
Termination Conditions

شروط إنهاء التنفيذ:

- يمكن تحديد نهاية التنفيذ في اللحظة t إذا تحقق لدينا ما يلي:
- لدينا نهاية موضعية لكافة الإجراءات في اللحظة t .
- لا يوجد أية رسائل متبادلة أو يجري تبادلها بين الإجراءات في اللحظة t .

إنهاء التنفيذ باستخدام رسائل الإقرار

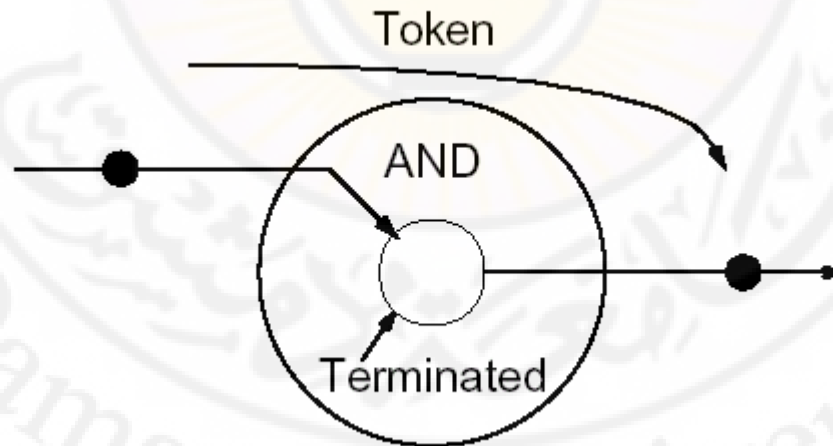
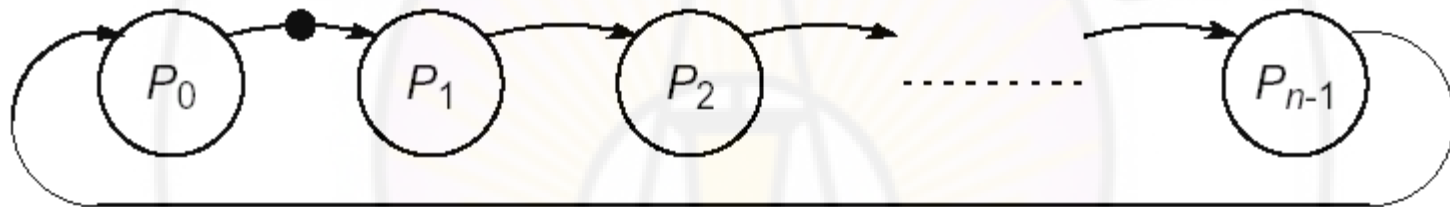
Termination Using Message Acknowledgments



إنهاء التنفيذ الحلقي

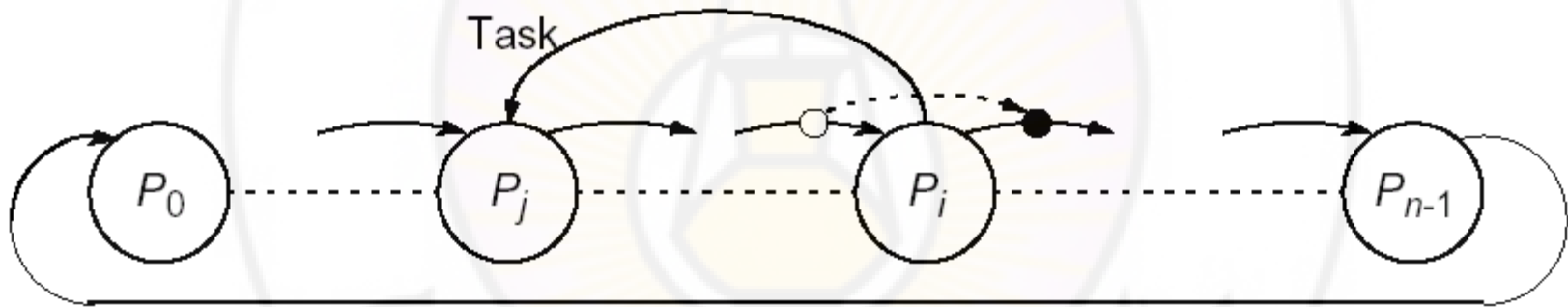
Ring Termination Algorithm

Token passed to next processor
when reached local termination condition



إنهاء التنفيذ الحلقي

Ring Termination Algorithm



إنهاء التنفيذ الشجري

Tree Termination Algorithm

