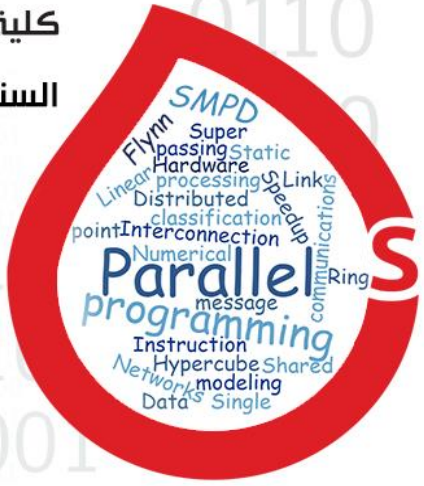
Amdahl low & Group messages



د. روان قرعوني

15/04/2024

RB Informatics;

البرمجة التفرعية

بسم الله الرحمن الرحيم

تعرفنا في المحاضرة السابقة على أنواع الحواسب التفرعية وميزاتها وعيوبها، وطريقة تصنيف البرنامج التفرعي حسب Flynn classification وطرق ربط الـ computers، وحساب زمن التحسين عن البرمجة التسلسلية.

سنتعرف في هذه المحاضرة على:

- قانون Amdahl للاستجابة العظمى
- عملية إرسال واستقبال الرسائل في التنفيذ المتزامن والغير متزامن
- Group messages

تذكرة: إن معامل الاستجابة هو: $\frac{ts}{tp}$

قانون Amdahl للاستجابة العظمى

- سؤال: هل كلما زاد عدد المعالجات في البرمجة التفرعية ينقص زمن التنفيذ؟
- مثلاً إذا كان لدي برنامج يُنفذ على عشرة معالجات هل إضافة عشرة معالجات أخرى ستجعل زمن التنفيذ أقل؟!
- هل نستطيع زيادة عدد المعالجات بشكل كبير؟

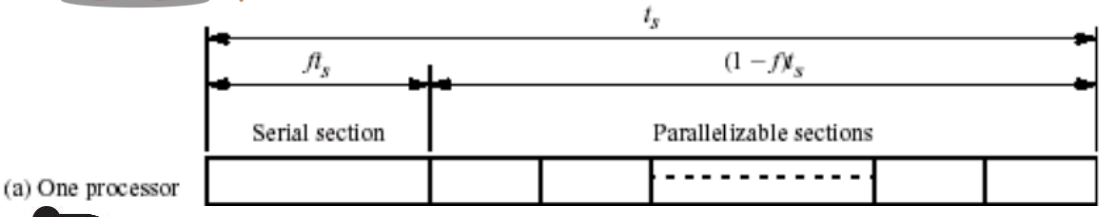
- الجواب هو **لا**، يوجد حد أعظمي للاستجابة، بعد تجاوزه تكون زيادة عدد المعالجات غير فعالة
  بحسب قانون Amdahl فإنه يوجد دائما في كل برنامج جزء تسلسي (يجب تنفيذه على التسلسل)

"ذكرنا سابقا أنه ليس جميع المسائل قابلة للتفرع بشكل كامل"

ليكن لدينا برنامج معين ولنقم بتمثيل الزمن اللازم لتنفيذه باستخدام كل من:

1. البرمجة التسلسلية t_s
2. البرمجة التفرعية $(ft_s \;\&\; (1-f)t_s)$

فيكون لدينا الشكل التالي:



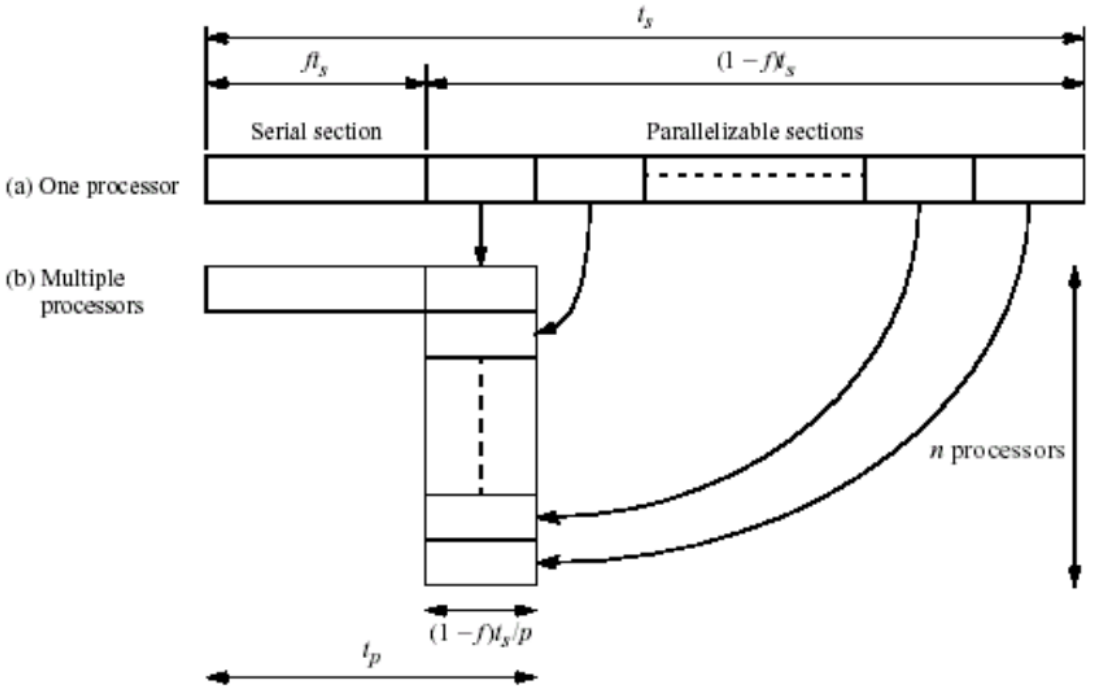
- نلاحظ أنه عند استخدام البرمجة التفرعية، يقسم وقت التنفيذ إلى:
  - قسم متسلسل غير قابل للتفرع (Serial section)
  - قسم قابل للتفرع (Parallelizable sections)

حيث f هي نسبة التسلسل في البرنامج ونجد أيضاً أنه:

- $f * ts$ هي زمن تنفيذ الجزء التسلسلي
- $(1 - f) * ts$ هي زمن تنفيذ الجزء غير المتسلسل (القابل للتفرع)

نستطيع توزيع القسم غير المتسلسل على المعالجات

بعد التوزيع على معالجات يصبح لدينا الشكل b



يصبح زمن التنفيذ للجزء القابل للتفرع هو

$$\frac{(1-f) * ts}{p}$$

وزمن لتنفيذ التفرعي الكلي هو:

$$tp = f * ts + \frac{1 - f * ts}{p}$$

لدينا قانون الاستجابة

$$S(p) = \frac{ts}{f * ts + \frac{(1-f) * ts}{p}}$$

بعد القيام بتوحيد المقامات يصبح

$$\frac{p * ts}{p * f * ts + (1-f) * ts}$$

نقوم بإخراج ال t_s ك عامل مشترك واختصارها من البسط والمقام يصبح

$$= \frac{p}{p * f + (1 - f)}$$

الآن نقوم بإخراج الـ p كعامل مشترك

$$= \frac{1}{f + \frac{1}{p} - \frac{f}{p}}$$

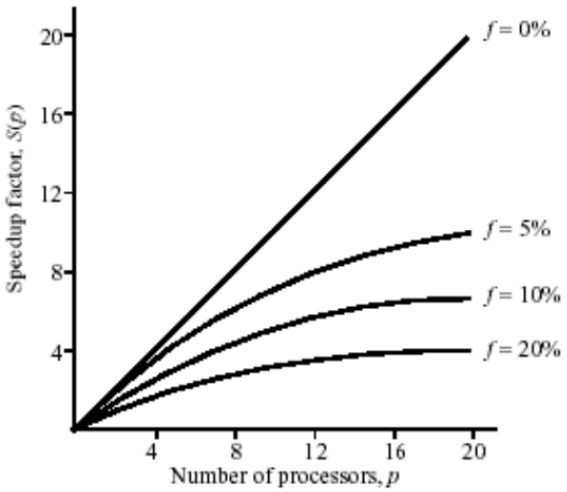
ولدينا القانون الرياضي:

$$\lim_{p \to \infty} \frac{1}{p} = 0$$

بتطبيق القاعدة على قانون الاستجابة

$$\lim_{p \to \infty} s(p) = \frac{1}{f}$$

- نجد أنه عندما يكون عدد المعالجات يسعى إلى قيمة كبيرة تكون الاستجابة مساوية لمقلوب الجزء التسلسلي من البرنامج ومنه نلاحظ أن نسبة التسلسل بالبرنامج هو العامل الأساسي في حساب الاستجابة العظمى
- نلاحظ العلاقة بين زيادة عدد المعالجات والاستجابة في المخطط التالي على اليسار:



- نلاحظ أنه عندما $f = 0$ (لا يوجد تسلسل بالبرنامج) ويسعى الخط للانهاية
- هل يعني هذا أنه في حالة عدم وجود جزء تسلسلي، كلما ازداد عدد المعالجات زادت السرعة بدون وجود حد لزيادة عدد المعالجات!؟
- الجواب: لا، تكون زيادة عدد المعالجات متعلقة أيضاً بطبيعة المسألة.

مثال للتوضيح

لدينا مصفوفة مربعة 4 * 4

ونريد زيادة كل قيمة في هذه المصفوفة بمقدار 1 لا يمكن استخدام اكثر من 16 معالج (لكل خانة معالج) وإلا تكون الزيادة غير مفيدة بينما في حال لدينا مصفوفة كبيرة جداً (مليار * مليار)

هنا يمكننا زيادة عدد المعالجات أكثر للوصول لاستجابة عظمى .

- الخلاصة : تزداد الاستجابة بزيادة عدد المعالجات إلى حد معين , يتعلق هذا الحد بكل من:
- نسبة التسلسل في المسألة
- طبيعة المسألة

زمن إرسال الرسائل (تبادل الرسائل)

في البداية يجب أن نتعرف على مفهومين أساسين وهما Static process creation وDynamic process creation

Static process creation

تكون المهام المطلوبة في المسألة معروفة قبل البدء بالتنفيذ فنقوم بخلق المهام المطلوبة في البداية ثم نقوم بإرسال الـ Data المطلوبة لكل مهمة وبعدها نقوم بجلب خرج كل مهمة

Dynamic process creation

في هذه الطريقة نقوم بخلق المهمة عند الحاجة فقط.

في القسم العملي سنتعرف على هذين المفهومين بشكل موسع وسنرى تطبيقها العملي لـ:

- Dynamic process creation (SPON) في الـ PVM
- Static process creation في الـ MPI

كما سنلاحظ انه غالبا تعد الـ static process creation كـ single program multiple data

وتعد الـ dynamic program creation كـ multiple program multiple data

بالعودة إلى إرسال الرسائل

يتم إرسال واستقبال الرسائل عن طريق التوابع Send / recv ويوجد نوعين للتنفيذ المتزامن والغير متزامن

1. في حالة التنفيذ المتزامن

تقوم كل تعليمة بانتظار مقابلتها حتى يتم استدعائها

توضيح:

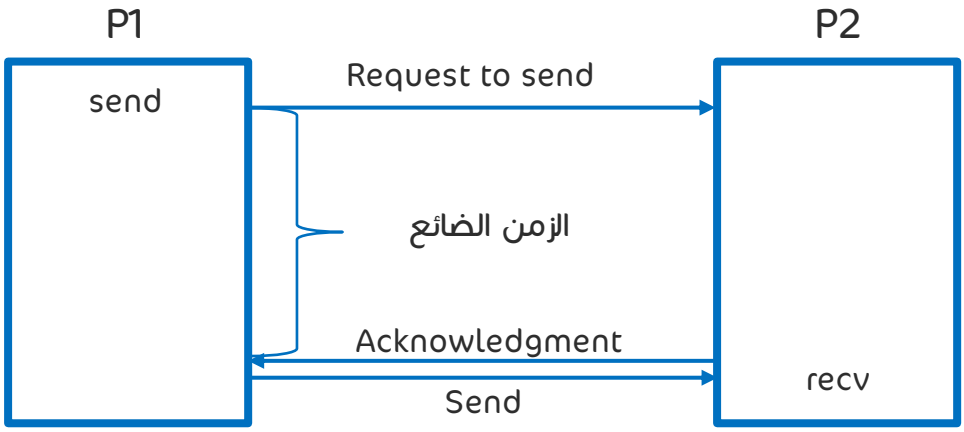
بداية يجب معرفة ان كل عملية تواصل بين المهام تتم باستخدام كلا التابعين

- Send
- recv

حيث نقوم باستدعاء كل تابع في مهمة

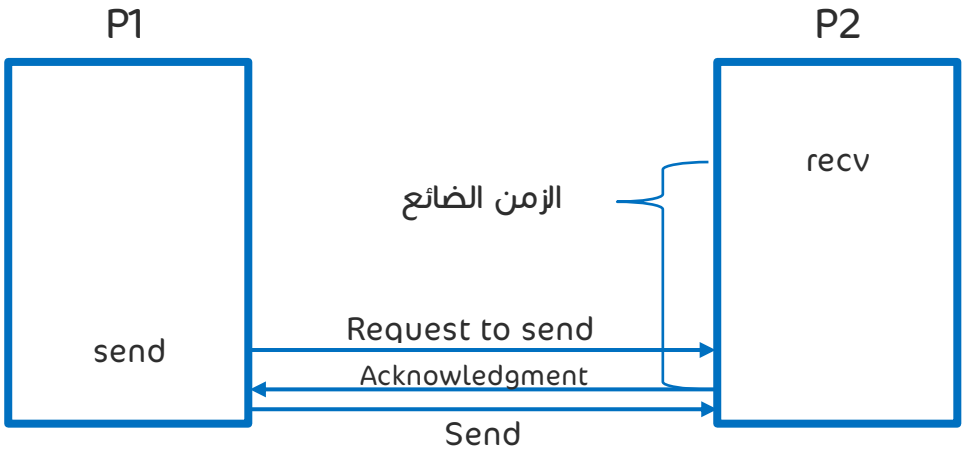
مثال: ليكن لدينا المهمتين P1 , P2

قامت العملية الأولى بتنفيذ التابع Send بإرسال "request to send" للعملية الثانية ويتوقف التنفيذ في العملية الأولى حتى تقوم العملية الثانية بإرسال الـ acknowledgment من خلال استدعاء تابع الـ recv ثم تقوم التعليمة الأولى بعملية الإرسال send وتستأنف التنفيذ



- بعبارة أبسط تقوم المهمة **الأولى** بطلب الموافقة على إرسال بيانات معينة، وتقوم بإيقاف التنفيذ في انتظار الموافقة من التعليمة **الثانية** على قبول هذه البيانات ثم يستأنف كليهما التنفيذ لكن نلاحظ وجود زمن ضائع بين إرسال العملية الأولى لطلب الإرسال وقبول التعليمة الثانية له.

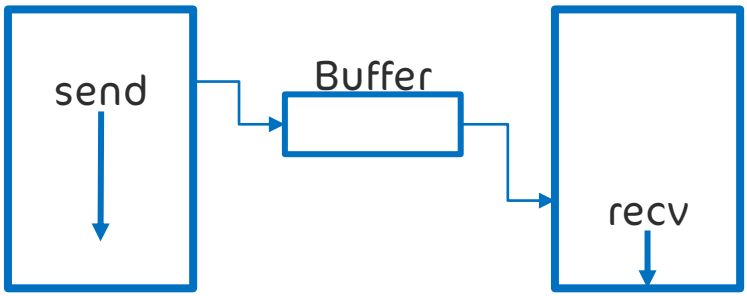
- ماذا يحدث في حال قمنا باستدعاء التابع recv في المهمة الثانية قبل استدعاء تابع send في المهمة الأولى؟
- سيتوقف التنفيذ في العملية الثانية حتى وصول طلب الإرسال من العملية الأولى، بعدها تقوم العملية الأولى بإرسال البيانات، ويستأنف كل من العمليتين التنفيذ وبسبب ذلك نلاحظ وجود زمن ضائع



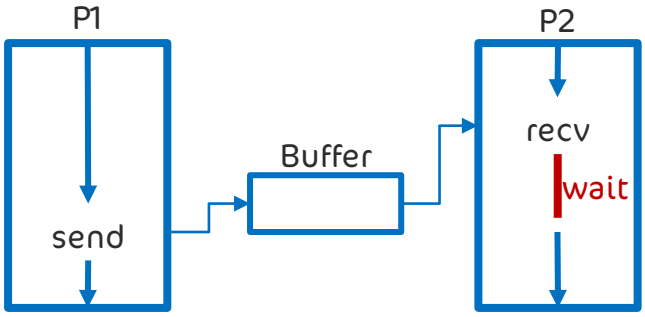
- **ملاحظة هامة:** يجب مقابلة كل تابع recv بتابع send وذلك لأنه في حالة وجود أحدهما سيتوقف التنفيذ في العملية الحالية حتى يتم تنفيذ التابع المقابل في عملية أخرى وفي حال عدم وجوده فإن التنفيذ لن يستأنف

2. في حالة التنفيذ الغير متزامن

لا تقوم المهام بإيقاف التنفيذ بعد استدعاء أحد التابعين (Send / recv) **أي أن تعليمة الـ** send **تقوم بإرسال الرسالة ويستأنف التنفيذ مباشرة، دون انتظار الـ** acknowledgment **من العملية الثانية لكن هنا تظهر الحاجة إلى تخزين الرسالة، لذلك سنستخدم** buffer



- سؤال: ماذا يحدث بحال إرسال عدة رسائل باستخدام send دون تلقي هذه الرسائل، ثم قامت عملية أُخرى باستخدام التابع recv لاستقبال رسالة معينة، كيف تميز الرسالة الصحيحة من الـ buffer؟
  - يكون لكل رسالة الـ tag الخاص بها، وهو أحد الـ parameters الموجودة في كل من التابعين Send / recv
- سؤال: ماذا يحدث في حال تم تنفيذ اكتر من أمر send حتى امتلاء الـ buffer؟
  - يتحول التنفيذ من غير متزامن إلى متزامن أي ان كل تعلمية تعود لانتظار التعليمة المقابلة لها
- سؤال: لنفرض ان العملية الأولى قامت ب تنفيذ التابع recv قبل إرسال أي رسالة (حيث نجد أن الـ buffer فارغ)؟
  - في هذه الحالة تتابع العملية الأولى التنفيذ مع التذكر بوجود رسالة لم تُستقبل بعد.
- ماذا لو كان التنفيذ معتمد على هذه القيمة الغير مستلمة بعد !؟
  - في هذه الحالة يقوم المبرمج باستخدام التعليمة wait لإعلام العملية بوجود قيمة يجب استلامها قبل متابعة التنفيذ



نلاحظ أنه بحال كانت القيمة قد استلمت فإنه سيتم تجاهل التعليمة wait

- ماذا يحدث في حال تجاهل المبرمج لاستخدام التعليمة wait والبرنامج بحاجة للقيمة الغير مستلمة ؟
  - يقوم البرنامج باستخدام قيم خاطئة مما يؤدي للحصول على نتائج خاطئة

Your work is going to fill a large part of your life, and the only way to be truly satisfied is to do what you believe is great work, And the only way to do great work is to love what you do...
If you haven't found it yet, keep looking...
Don't settle as with all matters of the heart, you'll know when you find it...
Steve Jobs...

Group message

في القسم العملي سنجد أن كل رسالة سنقوم بإرسالها بحاجة إلى عملية "تهيئة" قبل الإرسال، وكل عملية "تهيئة" تتطلب زمن وفي حال وجود عدد كبير من الرسائل سيصبح لدي زمن ضائع لذلك نقوم بجمع المهام (الأبناء) في مجموعة "group" وتهيئة الرسالة مرة واحدة وإرسالها.

يوجد 5 أنواع لعملية إرسال الرسائل ل group

1. Broadcast

وهي عملية إرسال الرسالة إلى جميع المهام في المجموعة ومن ضمنهم العملية التي قامت بالإرسال (ترسل إلى نفسها)

2. Multicast

نقوم بتحديد عدد معين من المهام التي سوف تستلم الرسالة.

تتم عملية تحديد المهام التي سوف تستلم الرسالة من خلال تحديد ال id لكل مهمة مُشاركة

3. Scatter

مثلا لديك مصفوفة من أربع خانات ولدينا مثلاً ضمن المجموعة أربعة مهام نقوم بتحديدهم (من ضمنهم المهمة التي ستقوم بإرسال الرسالة) وليكن الهدف هو زيادة كل قيمة من قيم المصفوفة بمقدار 2....

تقوم scatter بتهيئة الرسالة مرة واحدة وإرسالها، لكن **كيف تقوم باقي المهام باستقبال هذه الرسالة؟**

- تقوم المهام في نفس المجموعة باستقبال الرسالة باستخدام تابع scatter أيضا، أي أن كل من عملية الإرسال والاستقبال تتم من خلال تابع scatter

ملاحظات:

- تابع الـ scatter يحوي على العديد من الـ Parameters التي تتيح التحكم بالمصفوفة المرسلة، وكيفية توزيعها على الأبناء وحجم الـ Block المرسلة لكل إبن
- يمكن استخدام تابعي send/ recv بدلاً من تابع الـ scatter ولكن يستخدم لتوفير الوقت فبدلا من تهيئة الرسالة عدة مرات يقوم الـ scatter بتهيئتها مرة واحدة.

- ما الفرق بين scatter وmulticast ؟
- Multicast: لدينا قيمة واحدة، نقوم بتوزيعها نفسها لجميع المهام في نفس المجموعة،
- لدينا مصفوفة من القيم، نقوم بتوزيع كل قيمة إلى مهمة في المجموعة
- هل يجب أن تكون المهمة المُرسلة عضواً في ال group؟

نعم .

- ماذا لو كانت المصفوفة بطول خمس عناصر، وكان عدد المهام في الـ group أربعة؟
  تعد هذه الحالة خطأ من المبرمج، وفي حال تم الإرسال فإن القيمة التي سوف ترسل إلى المهمة الإضافية في الـ group تكون null

4. Gather

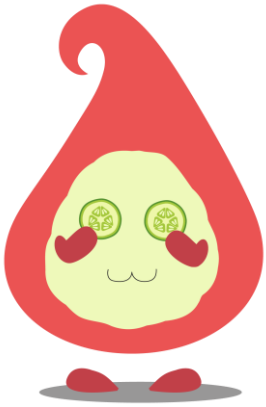
في هذا النوع يكون لدينا عدة مهام تقوم بإرسال رسائل لمهمة معينة، وتقوم هذه المهمة بتجميع هذه الرسائل في مصفوفة حيث يتم الإرسال والاستقبال باستخدام التابع gather

5. Reduce

كما في Gather، يكون لدينا مهمة واحدة تقوم باستلام الرسائل من المهام الأخرى في نفس المجموعة، إضافة إلى أنها تقوم بعملية معينة على الرسائل المستلمة، لتحول القيم الواردة إلى قيمة واحدة .

العملية قد تكون : ..... and , or , sum

- ملاحظات:
- مقابلة الـ reduce هو عدة استدعاءات لتابع الـ send
- تابع Scatter هو تابع توزيع
- تابع Gather هو تابع تجميع
- هذا لايعني أن الـ scatter للإرسال فقط والـ Gather للاستقبال
- في حال قررنا استخدام توابع scatter و Gather معاً سنحتاج:
  Scatter بالـ root وscatterg بالأبناء لاستقبال رسائل من تابع scatter
  ثم تابع Gather بالـ root و Gather بالأبناء لتجميع الرسائل من توابع Gather
- - ليس من الضروري أن كل scatter يقابلها Gather
  فمن الممكن استخدام توابع scatter لتوزيع المهام وعند الاستقبال نستخدم توابع send وrecvg
- ستتوضح استخدامات هذه التوابع أكثر خلال محاضرات العملي
- التوابع الخمسة السابقة هي توابع تجميعية تطبق على group كامل ويمكننا الاستغناء عنهم بتوابع send & receive وعندها سنعود لتهيئة الرسائل عند كل مهمة وزيادة الوقت الضائع



The End...

Love all, trust a few, do wrong to none." – William Shakespeare