نموذج تخزين /استرجاع لتطبيقات الحوسبة الشبكية بالاعتماد على سرقة العمل

رهف غزال و سمير جعفر

كلية العلوم – قسم الرياضيات – جامعة دمشق

الملخص

يندرج البحث في مجال التسامح مع الأعطال في البيئات التفرعيـــة الواسعـــة مثل الحوسبة الشبكية (grid) وعناقيد الحواسيب (cluster)، بهدف إيجاد أفضل الطرق للتعامل مع الأخطاء المتعلّقة بتعطل أحد الأجهزة الموجودة في البيئة أو الناتجة عن انقطاع شبكة الاتصال وذلك لضمان استمرارية عمل التطبيقات المتوازية المنفذة ضمن هذه البيئة في ظل وجود الأعطال.

قمنا في البحث بدراسة لنموذج البيئة التفرعية المعتمَد والتطبيقات المتوازية المنفذة ضمنه، ثم قدمنا آلية تخزين/ استرجاع تمكننا من ضمان استمرارية التطبيق في حال ظهور أي عطل باستخدام التمثيل المجرد لحالة التطبيق على المعالجات والمتمثّل بمخطط تدفق البيانات (macro dataflow) للتطبيقات التي تستخدم خوارزمية سرقة العمل (work stealing) لتوزيع المهام بين المعالجات وتُنفَّذ في بيئات تفرعية واسعة غير متجانسة وديناميكية، وذلك بكلفة بسيطة مضافة لكلفة التنفيذ المتوازي نتيجة حفظ جزء من العمل خلال التنفيذ الطبيعي (fault-free execution)، بالإضافة إلى تقديم نموذج رياضي لحساب التعقيد الزمني(الكلفة) لهذه الآلية المقترحة .

الكلمات المفتاحية : الحوسبة الشبكية، مخطط تدفق البيانات، سرقة العمل، التسامح مع الأعطال، نقطة تحقق، البرمجة المتوازية.

A checkpoint/recovery Model based on work stealing for grid applications

Rahaf Ghazal , Samir Jafar

Department of Mathematics – Faculty of Science

Damascus University

Abstract

The study is researching the fault tolerance in the large distributed environments such as grid computing and clusters of computers in order to find the most effective ways to deal with the errors associated with the crash one of the devices in the environment or network disconnection to ensure the continuity of the application in the presence of the faults. In this paper we study a model of the distributed environment and the parallel applications within it. Then we provide a checkpoint mechanism that will enable us to ensure continuity of the work used by a virtual representation of the application (macro dataflow) and suitable for the applications which uses work stealing algorithm to distribute the tasks which are implemented in heterogeneous and dynamic environment.

This mechanism will add a simple cost to the cost of parallel execution as a result of keeping part of the work during fault-free execution. The study also provides a mathematical model to calculate the time complexity i.e. the cost of this proposed mechanism.

Key words : grid computing, macro data flow, work stealing, fault tolerance, checkpointing, parallel programming.

1. المقدمة :

ظهرت تقنية الحوسبة الشبكية (grid) للحسابات المتوازية كأداة فعّالة لحل المسائل الكثيفة التي تحتاج إلى وقت حساب طويل ضمن حاسوب تقليدي واحد، حيث تعدّ بيئة الحوسبة الشبكية بشكلها العام عبارة عن تشارك مجموعة من الأنظمة الحوسبية المختلفة التي تتصل مع بعضها البعض عن طريق شبكة اتصال واسعة، يهدف هذا الربط إلى حل التطبيقات والمسائل الحسابية المعقدة بشكل أسرع، حيث نسمح لأكثر من معالج أن يشارك في الحل من موقعه الخاص ونستفيد من الموارد والخدمات التي يوفرها كل منهم من أماكن تخزينية وقدرات معالجة دون تحمُّل عبء استخدام حواسيب عملاقة لحل تلك التطبيقات، تتصف بأنها بيئة حسابية موزعة وديناميكية وذات مكونات غير متجانسة وتتصل فيما بينها عن طريق تبادل الرسائل (messages passing) [8] .

رافق هذه البنى ظهور الأعطال الكثيرة المتعلقة بمكوناتها سواء أعطال العقد (المعالجات) أو أعطال الشبكة (انقطاع الاتصال)، وبالتالي فإن الخدمات المقدمة من قبل المستخدمين في التطبيق قد تفشل ولا يستطيع التطبيق أن يتم عمله بشكل ناجح [6],[7]، فتم اقتراح آليات نزود بها نظام الحوسبة الشبكية بحيث تصبح مكوناته قادرة على التعامل مع هذه الأحداث، وذلك دون تحميلها عبء على الأداء أو زيادة كبيرة في وقت تنفيذ التطبيق المتوازي، فتم دمج مفهومي : تحديد العطل وإعادة تشغيل التطبيق "recovery" ضمن مفهوم التسامح مع الأعطال " fault tolerance" من أجل التطبيقات المتوازية [1],[2] ،وهنا اتجهت الدراسات لإيجاد أفضل الخوارزميات حتى تكون عملية الاستعادة فعّالة ويُستفاد منها في تجاوز مرحلة الفشل بكلفة خسارة مقبولة .

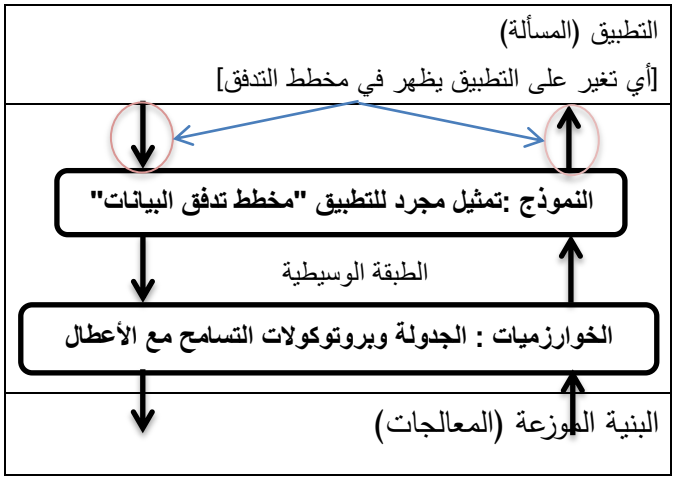
2. هدف البحث:

يهدف العمل إلى إيجاد آلية جديدة فعالة للتسامح مع الأعطال ضمن بيئة حسابية موزعة ذات مكونات غير متجانسة وديناميكية للتطبيقات المتوازية كما في بيئة الحوسبة الشبكية (grid computing) وعناقيد الحواسيب (computer cluster) حيث سيُستخدم مبدأ سرقة العمل لتوزيع المهام بين المعالجات ومخطط تدفق

البيانات (macro data flow) لتمثيل حالة تنفيذ التطبيق بشكل مجرد، علماً أنّ المهام غير مستقلة (مرتبطة)، سعياً لأن تكون كلفة التخزين خلال التنفيذ الطبيعي أقل ما يمكن (تسعى إلى الصفر) وفي حال وجود حدث استثنائي خلال العمل – حدوث عطل ما – سنكون على استعداد لتحمل كلفة الخسارة الناتجة على أن تكون محدودة لضمان استمرار عمل التطبيق وذلك بفرض أن عدد الأعطال خلال فترة التنفيذ قليلة جداً بالنسبة لعدد المعالجات المشاركة.

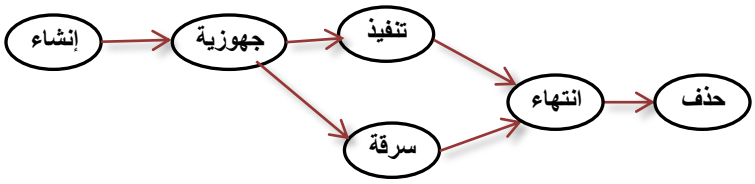
3. مواد وطرق البحث :

- شكلت البيئات التفرعية الواسعة مثل الـ grid تحدياً لدى الباحثين من أجل المحافظة على موثوقيتها وجودة عمل التطبيقات ضمنها مع زيادة عدد المكونات المشاركة والتي يمكن ان تتعطل، وبذلك ظهرت آليات التسامح مع الأعطال (fault tolerance) وهي عبارة عن مجموعة من التعليمات التي يتم تزويد مكونات الشبكة بها للتعامل مع الأعطال المختلفة في حال ظهورها .
- بيئة التنفيذ المتوازية المعتمدة هي KAAPI وهي اختصار لـ (Kernel for Adaptive Asynchronous Parallel Interface) [5] تتألف من ثلاث مستويات، مستوى التطبيق ومستوى البيئة الموزعة المادية وبينهما مستوى وسيطي نطبق ضمنه خوارزميات الجدولة وخوارزميات التسامح مع الأعطال وأيضا نمثل فيه التطبيق باستخدام بنية افتراضية مجردة هي مخطط تدفق البيانات (Macro Data Flow).
- سبب اختيار ما يسمى الطبقة الوسيطية هو إمكانية إيجاد خوارزميات تُطبَّق بشكل مستقل عن نوع المسألة المراد حلها ضمن البيئة وعن نوع وطبيعة الأجهزة المشاركة في الحل وشفافة بالنسبة للمستخدم (transparency) .
- ينفذ التطبيق بأكمله على p معالج فيزيائي(processor) وكل من هذه المعالجات الفيزيائية يتكون من عدة معالجات خفيفة (processes) وهذه الأخيرة لديها عدة مسالك (threads) ويتم تنفيذ العمليات عليها بشكل تسلسلي .



الشكل 1: المخطط العام للنموذج البرمجي للتطبيق المتوازي في الـ KAAPI

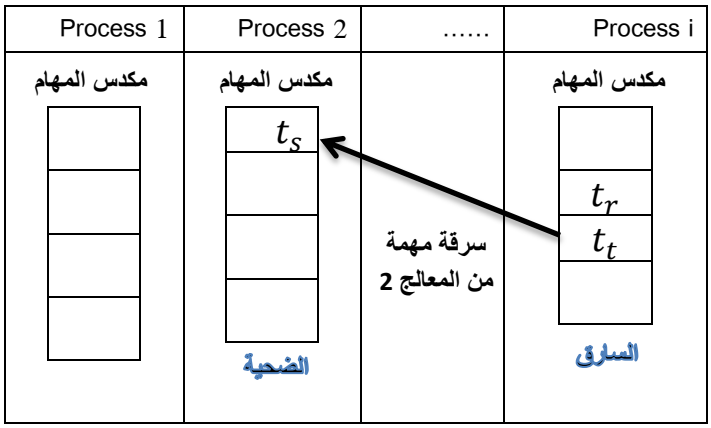
- مخطط تدفق البيانات (macro data flow)[5]: يعتبر تمثيل مجرد للتطبيق المتوازي بصيغة محمولة (portable) وهو بيان موجه $G=(V, \Sigma)$ حيث V مجموعة منتهية من الرؤوس وتمثل المهام (tasks) الحسابية بالإضافة للمعطيات المشتركة بين المعالجات (shared data) ، و Σ مجموعة الأضلاع وتمثل التبعية بين المهام، ويتميز هذا المخطط بأنه ديناميكي أي أنه ينشأ ويتغير أثناء التنفيذ تبعا لتشكل مهام جديدة أو حذف أخرى منتهية ، وفيما يلي سنعرض الحالات التي تمر بها أي مهمة في مخطط التدفق :



الشكل 2 : دورة حياة المهمة في مخطط تدفق البيانات

- سرقة العمل (work stealing) [4]: هي خوارزمية جدولة تقوم بتوزيع عبء العمل على المعالجات أثناء التنفيذ، ومبدؤها يقوم على أن أي معالج خفيف (process) خامل (idle) يستطيع سرقة مهمة جاهزة من معالج خفيف آخر نطلق عليه اسم

الضحية (victim) ونطلق على الأول اسم السارق (thief)، حيث أنه وعند السرقة ينشئ المعالج السارق مهمة السرقة t_t التي ترسل طلب السرقة إلى الضحية وعندما يجد الضحية مهمة جاهزة[1] في قمة مكدسه مثل المهمة t_s فإنه يسمح للسارق بسرقتها وعندما ينتهي السارق من تنفيذها فإنه ينشأ المهمة t_r التي تكون مسؤولة عن إعادة النتيجة إلى الضحية كما هو موضح في المخطط التالي:



الشكل 3 : نموذج المعالج (processor) في بيئة الـ KAAPI وإظهار عملية السرقة بين المعالجات الخفيفة

4. آلية التسامح مع الأعطال المقترحة :

تعتمد الآلية على تخزين جزء من معلومات التطبيق خلال التنفيذ الطبيعي، ومن ثم استرجاعها لاحقا في حال وقوع عطل ما لأحد المكونات، كما يلي:

i. مرحلة التخزين (checkpointing) :

ويتم فيها حفظ المهام المسروقة ونتائجها خلال التنفيذ الطبيعي باستخدام رتل سرقة (steal queue) وتسجيل حالة التطبيق المنفذّة على المعالج الرئيسي باستخدام نقاط تحقق دورية كما هو مبين فيما يلي:

[1] تكون المهمة جاهزة عندما تتوفر جميع مدخلاتها.

رتل سرقة (steal queue) ويستخدم لتسجيل التبعية بين المعالجات الخفيفة ، فعند حدوث سرقة بين معالجين ننشئ للسارق رتل سرقة نحفظ فيه المهمة المسروقة، ونخزّن هذا الرتل ضمن ذاكرة مستقرة (stable memory).

جدول النتائج العام (global result table) ويكون موجود لدى كل المعالجات المشاركة حيث تُحدَّث معلوماته أثناء التنفيذ وتذاع على المعالجات باستخدام أسلوب الـ broadcast، نستخدمه لحفظ نتائج الأعمال الأيتام بعد العطل (الخطأ) .

أما المعالج الرئيسي (master) فإننا نجعله يأخذ *نقاط تحقق محلية دورية (local periodically checkpoint)* لحفظ حالة التطبيق المنفذة عليه ضمن ذاكرة مستقرة بدور نرمز له بـ τ [2].

يقوم أي معالج سارق بإنشاء رتل سرقة خاص به يحفظ ضمنه المهام التي سرقها من معالجات أخرى بالشكل التالي:

(id of task, id of victim, parameter**1**,parameter**2**,…,parameter**m**, result)

" تعبر الوسطاء عن مجموعة مدخلات المهمة ".

وأما نتائج المهام الأيتام فتُسجَّل في جدول النتائج العام بالشكل التالي :

(id of task , id of process , result)

ii. مرحلة الاسترجاع (recovery) :

عند تعطل معالج ما وانقطاعه عن الشبكة لسبب ما يتم تعميم العطل على جميع المعالجات ونميز حالتين:

a) المعالج المعطّل هو معالج عادي :

يحل محل المعالج المعطّل معالج بديل يحمل نفس المعرّف الخاص بالمعالج المعطّل[3]، فيقوم بالبحث ضمن رتل السرقة المخزّن عن مهام مسروقة وغير مكتملة لكي يتمَّها فإن لم يجد يُعامَل كأنه عقدة جديدة ويحاول سرقة مهمة جاهزة من أحد المعالجات الأخرى .

[2] نختار هذا الدور بحيث يساوي الوقت الأعظمي لتنفيذ مهمة ما .

[3] في حال عودة المعالج المعطل إلى العمل فإنه يستطيع أن يشارك بالحل ولكن بمعرف جديد.

أما المعالج الذي يجد ضمن رتل سرقته مهمة قد سرقها من المعالج المعطّل (orphan task[4]) فإنه يتوقف عن إتمامها ويذيع النتائج الجزئية أو الكلية إلى جميع المعالجات الحية (live) وتُحفظ ضمن جدول النتائج العام الموجود لدى جميع المعالجات ، ليُعاد استخدامها في وقت لاحق عند الاسترجاع عندها يصبح المعالج السارق خامل ويحاول سرقة مهمة جاهزة جديدة من معالج آخر.

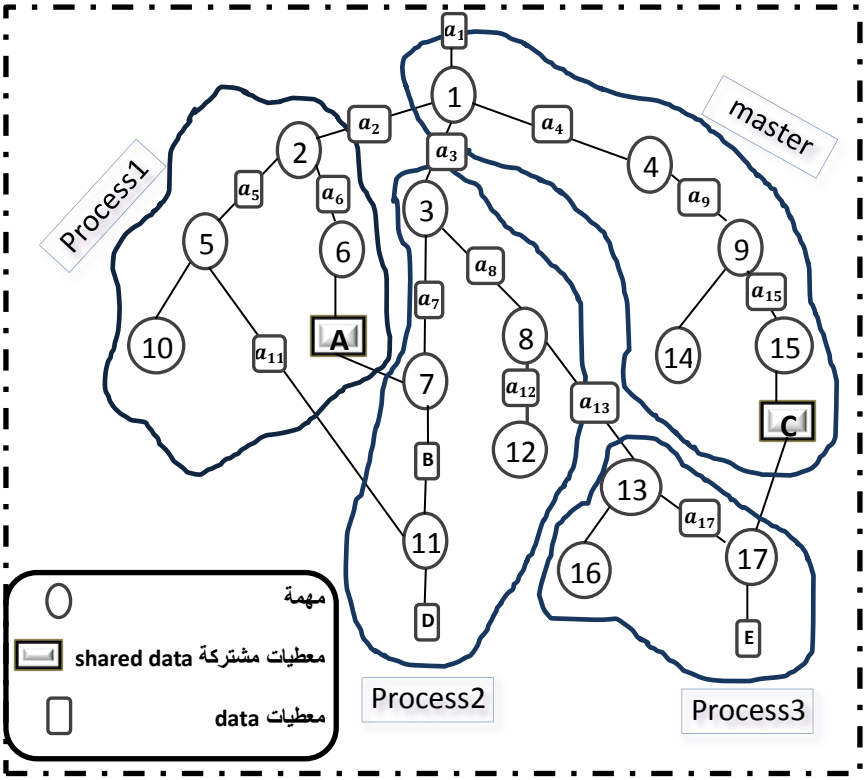
b) المعالج المعطّل هو المعالج الرئيسي :

في حال كان المعالج المعطل هو المعالج الرئيسي فإن الاسترجاع يتمثل بمتابعة تنفيذ التطبيق على معالج رئيسي بديل، بالاستعانة بآخر نقطة تحقق محفوظة لذلك المعالج المعطّل .

5. مثال توضيحي :

نعرض فيما يلي مثالاً عن تطبيق ما، يُنفَّذ بشكل متوازي ومُمثَّل باستخدام مخطط تدفق البيانات، نناقش فيه كل من مرحلتي التخزين والاسترجاع في حال حدث خطأ أثناء التنفيذ كما يلي:

[4] المهمة اليتيمة هي المهمة المسروقة من معالج معطّل.



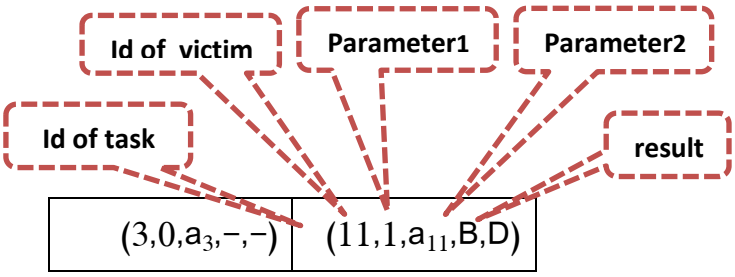
الشكل 4 : مثال عن مخطط تدفق بيانات *macro data flow*
خلال التنفيذ الطبيعي *free fault execution*

يعرض لنا الشكل 4 حالة تنفيذ تطبيق ما على أربعة معالجات وعمليات السرقة التي تحدث في لحظة ما أثناء التنفيذ، عندها وفقاً للخوارزمية المقترحة تتشكل أرتال السرقة الخاصة بكل معالج سارق وتُحفظ في ذاكرة مستقرة، مثلاً نعبّر عن رتل السرقة الخاص بالمعالج 2 كما هو موضح في الجدول1 :

Id of task	Id of victim	Parameter1	Parameter2	Result
3	0	a_3	-	-
11	1	a_{11}	B	D

الجدول 1 : رتل السرقة الخاص بـ *process2*

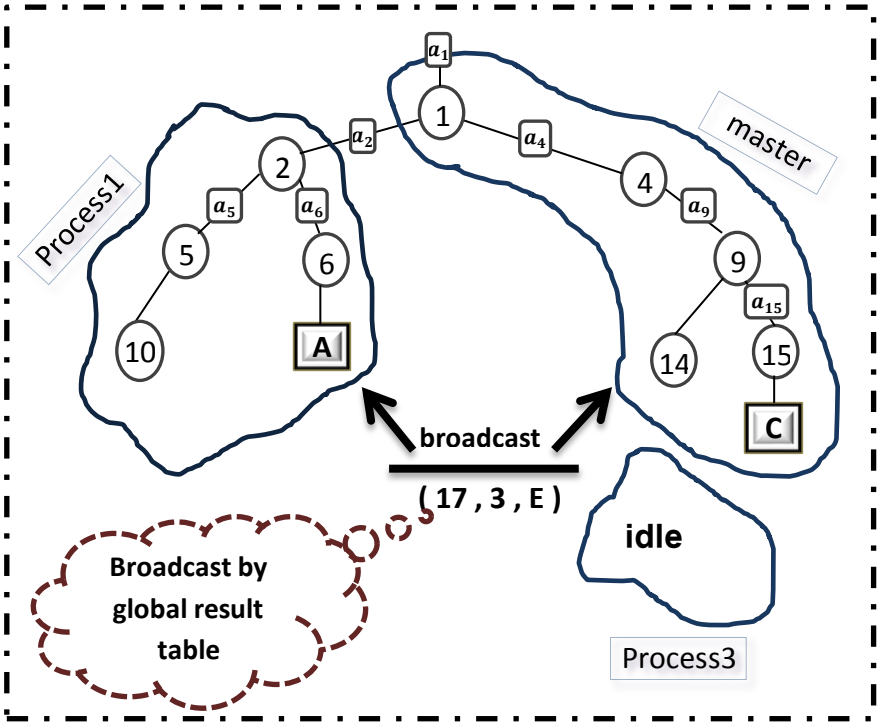
أو ممكن أن نعبّر عن رتل السرقة المتعلّق بالمعالج 2 بالطريقة :



الجدول 2 : رتل السرقة الخاص بـ *process2*

"حيث الـ 0 تشير إلى رقم المعالج الرئيسي" .

لبيان مرحلة الاسترجاع عند ظهور خطأ ما نعرض المخطط التالي:



الشكل 5 : تعطل المعالج الخفيف 2

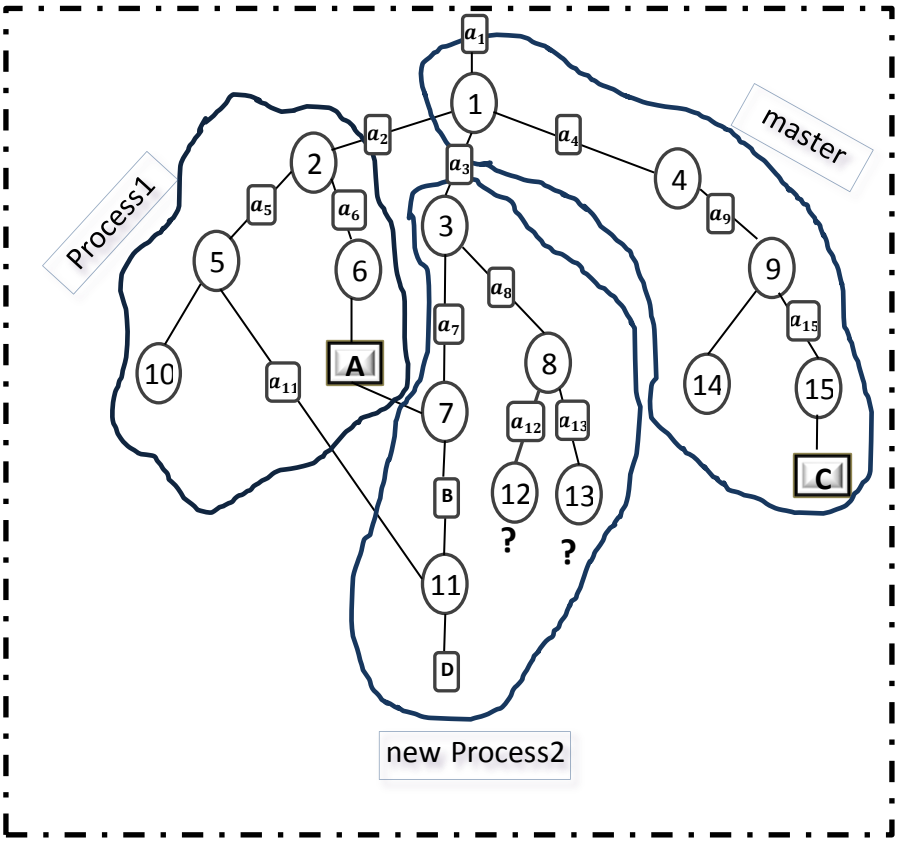
(crash of process2)

- في حالة تعطّل المعالج process2 في لحظة ما أثناء التنفيذ يعمم الخطأ على جميع المعالجات وتتم عملية إذاعة النتائج الجزئية (نتائج المهام الأيتام) التي توصل إليها المعالج process3 ضمن جدول النتائج العام على جميع المعالجات الحية (live) فيكون الجدول كما يلي:

Id of task	Id of process	Result
17	3	E

الجدول 3 : جدول النتائج العام

- ستقوم خوارزمية الاسترجاع بإعادة تشكيل مخطط تدفق البيانات الجزئي الخاص بالمعالج المعطّل من خلال قراءة رتل السرقة المخزّن حتى يكمل المعالج البديل عمله، حيث يبدأ الأخير (new process2) بإعادة تنفيذ المهمة 3 التي كان المعالج المعطّل القديم (process2) قد سرقها من المعالج الرئيسي فيتابع البديل تنفيذها، أما المهمة المسروقة 11 التي حُفظت نتيجتها فلا يُعاد تنفيذها وترسل نتيجتها إلى المعالج الأب (process1)، ماعدا ذلك فإن جميع المهام التي نُفذَّت على المعالج المعطّل القديم process2 قبل العطل سيُعاد إنشاؤها لعدم حفظ أي معلومات متعلقة بها من وسطاء أو نتائج مع التأكيد إلى أنه لا يوجد أي ضمان لإعادة التنفيذ بنفس الأسلوب الذي كان قبل العطل أي ممكن للمهام التي كانت قد نُفِّذت على المعالج 2 قبل العطل أن يُعاد تنفيذها على المعالج البديل أو أن تُسرق من معالجات أخرى لتُنفَّذ عليها.
- أما المعالج process3 فيصبح خامل بعد إذاعة نتائجه ويتم حذف المعلومات الموجودة لديه في رتل سرقته ويحاول سرقة مهمة جاهزة من معالج ما ، في هذه الحالة ممكن أن يعود المعالج process3 لسرقة المهمة 13 أو ممكن أن تُنفّذ على الـ new process2 دون سرقتها حيث يقوم process3 بسرقة مهمة أخرى، وفي كلا الحالتين يتم الحصول على نتيجة المهمة 13 الجزئية والتي تتمثل بنتيجة المهمة 17 المذاعة في جدول النتائج .



الشكل 6 : إعادة بناء مخطط تدفق البيانات الجزئي
للمعالج الفاشل *process2*

6. تحليل كلفة البروتوكول (الآلية) :

بداية لدراسة الكلفة نعرّف بعض المتغيرات المتعلقة بالتنفيذ المتوازي كما يلي:

t_m هو الزمن الوسطي لتخزين مهمة ما على ذاكرة مستقرة .

N_{theft} هو العدد الأعظمي للمهام المسروقة على معالج ما.

T_∞ يعبر عن زمن تنفيذ الخوارزمية المتوازية على عدد غير محدود من المعالجات بمعنى آخر زمن تنفيذ الجزء التسلسلي من البرنامج (المسار الحرج).

T_s زمن تنفيذ الخوارزمية التسلسلية على معالج واحد وهي تمثل كلفة طريقة الحساب التي يستخدمها التطبيق دون أي كلفة إضافية مرتبطة بالتوازي.

T_p زمن تنفيذ الخوارزمية المتوازية على p معالج فيزيائي .

T_1 زمن تنفيذ الخوارزمية المتوازية على معالج واحد، حيث يتألف من قيمة T_s مضروباً بكلفة التنفيذ المتوازي الممثل بكلفة تقسيم المهام (جدولتها) .

أي: $T_1 = c_1 \times T_s$

وبالتالي يمكن تنفيذ العمل المتوازي على p معالج باستخدام خوارزمية سرقة العمل لتنفيذ الجدولة بوقت متوقع هو [3]:

$$T_p = T_1/p + O(T_\infty)$$

فإذا عرَّفنا كلفة المسار الحرج بأنه ثابت صغير c_∞ ، فستصبح المعادلة السابقة بالشكل : $T_p \leq T_1/p + c_\infty T_\infty$

والأكثر من ذلك فإن نظرية الركود المتوازي (parallel slackness assumption) [3] تنص على : $T_1/p >> c_\infty T_\infty$

وبالتالي : $T_p \approx T_1/p$

في هذا البحث سوف نهتم بالتطبيقات القابلة للتوزيع أكثر ما يمكن أي التي يكون فيها $T_p >> T_\infty$.

i. التحليل خلال التنفيذ الطبيعي :

من ميزات هذا البروتوكول بأنه وعند التنفيذ الطبيعي فلا يوجد كلفة إضافية كبيرة للحفظ حيث يتم حفظ المهام المسروقة فقط ضمن ذاكرة مستقرة، إذاً تتحدد هذه الكلفة بالدالة $f_{overhead}(N_{theft}) = N_{theft} \times t_m$.

و بالاعتماد على مبدأ العمل أولاً (work first principle)[5] تتحدد العلاقة بين عدد السرقات وزمن تنفيذ المسار الحرج بالشكل : $N_{theft} = O(T_\infty)$ أي أن عدد السرقات صغير ومحدود بالـ T_∞.

[5] ينص على جعل كلفة جميع الاتصالات الناتجة عن الجدولة ، محدودة بزمن تنفيذ المسار الحرج [3].

فإذا كان T_p' يشير إلى التنفيذ المتوازي ضمن بيئة الـ KAAPI على p معالج وذلك باستخدام البروتوكول المقترح ، فإن :

$$T_p' = T_p + \text{cost of protocol (COP)}$$

حيث (COP) هي كلفة البروتوكول المقترح المضافة على البرنامج ، ومنه:

$$T_p' \leq T_p + (N_{theft} \times t_m)$$

وبالتالي من العلاقة السابقة نستنتج أن الكلفة المضافة إلى زمن التنفيذ المتوازي باستخدام آلية التخزين المقترحة تكون محدودة بعدد المهام المسروقة في التطبيق مضروباً بزمن وصول المهمة الواحدة إلى الذاكرة المستقرة (أكبر زمن ممكن) ، وكما فرضنا سابقا فإن عدد السرقات محدود بزمن تنفيذ المسار الحرج (T_∞) في التطبيق وأن $T_p >> T_\infty$ بالتالي فإن الكلفة المضافة صغيرة جداً ويمكن أن نكتب $T_p' \approx T_p$.

ii. التحليل عند حدوث عطل :

- عند تعطل المعالج الرئيسي :

عند تعطله في لحظة ما أثناء التنفيذ فإننا سنخسر على الأكثر مهمة واحدة (تبعاً لاختيار τ) وبالتالي الزمن الأعظمي لمرحلة الاسترجاع يتعلق بالزمن اللازم لتنفيذ مهمة واحدة والذي يكون محدود بـ τ.

- عند تعطل معالج عادي:

فإن خوارزمية الاسترجاع للمعالج المعطّل تنص على إعادة بناء مخطط تدفق البيانات الجزئي G_i الخاص بالمعالج المعطّل باستخدام رتل السرقة وجدول النتائج العام.

وبالتالي فإن كمية المهام التي سيُعاد تنفيذها ستكون على الأكثر تساوي $(N_1 - 1)$[6] أي في أسوء الحالات فإن المعالج سيكون قد سرق مهمة واحدة فقط وبالتالي عند الاسترجاع سيعيد تنفيذ جميع مهامه عدا المهمة المسروقة المنتهية وبالتالي الزمن الأعظمي اللازم لتنفيذ تلك المهام (نرمز له بـ t_i) يتعلق بقدرة البيان الجزئي G_i أي: $t_i = O(|G_i|)$

[6] N_1 تمثل العدد الأعظمي للمهام المنفذة على معالج ما.

7. الخاتمة والعمل المستقبلي:

قدمنا في هذا البحث دراسة لمفهوم التسامح مع الأعطال، توصلنا من خلالها إلى بروتوكول جديد للتسامح مع الأعطال للتطبيقات المتوازية المنفذة ضمن بيئات تفرعية واسعة، وذلك بالاستفادة من التمثيل المجرد للتطبيق (مخطط تدفق البيانات)، حيث يقوم البروتوكول على أساس حفظ المهام المسروقة خلال التنفيذ الطبيعي لاسترجاعها في وقت لاحق، وبالتالي تكون الكلفة المضافة إلى التنفيذ المتوازي محدودة بعدد المهام المسروقة في التطبيق وهو عدد محدود بزمن تنفيذ المسار الحرج في التطبيق، بالمقابل فإن الخسارة للمعلومات غير المحفوظة عند حدوث عطل ما لن تكون كارثية تؤدي إلى توقف التطبيق وستكون محدودة بالعدد الأعظمي للمهام التي يمكن تنفيذها على معالج ما.

وهنا ممكن أن تتجه الدراسة إلى البحث في إجابة السؤال التالي :

هل من المفيد زيادة كلفة التخزين مقابل خسارة أقل في الحساب عند وجود خطأ ما ؟

ومن هذا المنطلق فإن بروتوكول التسامح مع الأعطال المعتمد سيرجع إلى نوع التطبيق ومتطلبات المستخدم .

8. المراجع:

1. AVIZIENIS A, LAPRIE JC, and RANDALL B ,2001 ,Fundamental Concepts of Dependability, in University of Newcastle upon Tyne, Computing Science.
2. BALA A, CHANA I, 2012, Fault tolerance-challenges, techniques and implementation in cloud computing,in IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, No 1.
3. FRIGO M, LEISERSON CE, and RANDALL KH, 1998 ,The implementation of the Cilk-5 multithreaded language, in Proc. ACM SIGPLAN conference on Programming language design and implementation, Pages 212 - 223.
4. JAFAR S, KRINGS A , GAUTIER T, and ROCH JL , 2005, Theft-Induced Checkpointing for Reconfigurable Dataflow Applications ,in Proceedings of the IEEE Electro/Information Technology Conference .
5. JAFAR S, KRINGS A ,and GAUTIER T, 2009 ,Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing, in IEEE trans. Dependable sec. comput. 6(1), ,pages:32-44.
6. KAUR I, SINGH S, 2014, Detection of Crash Transient Failure during Job Scheduling using Replication Technique, in IJECS International Journal Of Engineering And Computer Science , Vol. 03 Issue 07 Pages 6904-6908.
7. NEOCLEOUS K, DIKAIAKOS MD, 2006, Grid Reliability: A Study of failures on the EGEE Infrastructure, in Proceeding of the CoreGRID Integration Workshop, pages : 165-176 .

8. PALMIERI F, PARDI S, 2010 , Towards a federated Metropolitan Area Grid environment: The SCoPE network-aware infrastructure, in Future Generation Computer Systems Vol. 26, Issue 8, Pages: 1241-1256.