Master Informatique Fondamentale, parcours Système Complexe

Termination Detection for HPC Task BASed environnement

Elève :
Paul Peseux

Cours :
Scheduling @ Scale

ENS DE LYON

Version 2.4 du
16 Novembre 2018

Contents

Introduction **1**

1 Context **2**

1.1 High-performance computing (HCP) . . . 2

1.2 Termination detection issue . . . 3

1.2.1 Why it is an issue . . . 3

1.2.2 Example . . . 3

2 Strong aspects **5**

2.1 Exisiting solution : HCDA . . . 5

2.2 Solution prposed : CDA . . . 5

2.3 Token ring . . . 6

2.4 Simulations . . . 7

2.5 Implementation . . . 8

3 Weak aspects **9**

3.1 C_{init} and C_{con} . . . 9

3.2 BORROW message . . . 9

3.3 Optimization . . . 9

4 Comparison with related work **11**

4.1 Other algorithms . . . 11

Conclusion **12**

Introduction

Here is a report of the article "Distributed Termination Detection for HPC Task BASed environnement" published by George Bosilca, Aurelien Bouteiller, Thomas Herault, Valentin Le Fèvre, Yves Robert and Jack Dongarra. It was published in June 2018 as a research report of the INRIA mathematics. It proposes a new algorithm to detect termination in High-performance computing. The algorithm, called Credit Distribution Algorithm (CDA), is an evolution of HCDA where modifications have been taught to improve performance in practical. An implementation of the CDA is performed by the author, in order to compare their results with state of the art of this termination detection issue.

Chapter 1

Context

1.1 High-performance computing (HCP)

High performance computing is the science of super-computer. The aim of this field is to achieve the highest performance possible in term of calculus with computers. The real point of this is to build computers that will be able to solve problems that requires a huge amount of calculus. For example we can cite meteorological predictions, climate study, chemical object simulation, physical simulations ... All these problems need high calculation because their models are quite complex, and the more is taken into account, the better the model. This High performance computers are ranked in TOP500, a ranking that exists since June 1993 and that is refreshed two times a year. An author of the article, Jack Dongarra is one of the responsible of this refreshing. The performance test is performed by the Linpack test that delivers a R_max score for every supercomputer. Here is an illustration of the leader R_max.

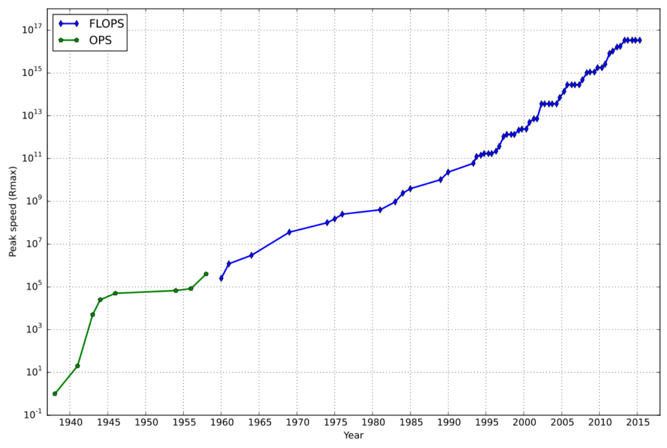

Figure 1.1: R_{max} evolution through time

1.2 Termination detection issue

1.2.1 Why it is an issue

We understood that HPC requires a lot of calculus. But once the computing is launched, it is critical to know when it stopped. Indeed we are dealing with a lot of different components. For example the actual leader of TOP500 is Sunway TaihuLight, located in China and has 40960 processors and 10649600 cores. As a consequence, as said in the article, if the application is dynamic, "no process has complete knowledge of the global configuration".

1.2.2 Example

Here we give an example and and implementation of such an application. Let's say we have a function f from $[0\,;1]$ to $\mathbb{R}$ which is C^∞.

Let's assume that we know f and f' through an oracle: we do not have a closed-form expression of f and f', but if we ask the oracle a value of f or f', it gives it to us.

Let's assume that f' is k-lipchitz, that is to say it exists $k \in \mathbb{R}$ such that

$$\forall x, y \in [0\,;1] \quad \mid f(x) - f(y) \mid \leq k \mid x - y \mid$$

The objective is to find a c-approximation g of f :

$$\forall x \in [0\,;1], \quad \mid f(x) - g(x) \mid \leq c$$

As a f' is k-lipchitz we have

$$\text{if} \mid f'(x) \mid \leq A, \forall y \in [x - \epsilon\,; x + \epsilon] \mid f'(y) \mid \leq A + k\epsilon$$

With $A = \frac{c}{\epsilon} - k\epsilon^2$, we have

$$\forall y \in [x - \epsilon\,; x + \epsilon] \mid f(x) - f(y) \mid \leq C$$

by simple intagration.

From these calculus, we derive a simple algorithm that gives a g, which is a c-approximation of f.

We construct g this way. From a finite set of intervals of $[0\,;1]$ of the form $\bigsqcup_{i \leq n} [a_i\,; b_i]$ we have a unique function that is constant on every $[a_i\,; b_i]$ with value $f(\frac{a_i + b_i}{2})$.

We start with $\bigsqcup_{i \leq 1} [a_i\,; b_i] = [0\,;1]$. Then for every interval $[a_i\,; b_i]$, if not $\mid f'(\frac{a_i+b_i}{2}) \mid \leq \frac{c}{\epsilon} - k\epsilon$, we split the interval in two intervals $[a_i\,; \frac{a_i+b_i}{2}]$ and $[\frac{a_i+b_i}{2}\,; b_i]$.

It is easy to prove that this algorithm converges as $A_\epsilon \underset{\epsilon \to 0^+}{\longrightarrow} \infty$ and as we work on a compact set.

Here is a Python implementation with $f = \cos x$, that is 1-lipchitz, with $c = 0.1$:

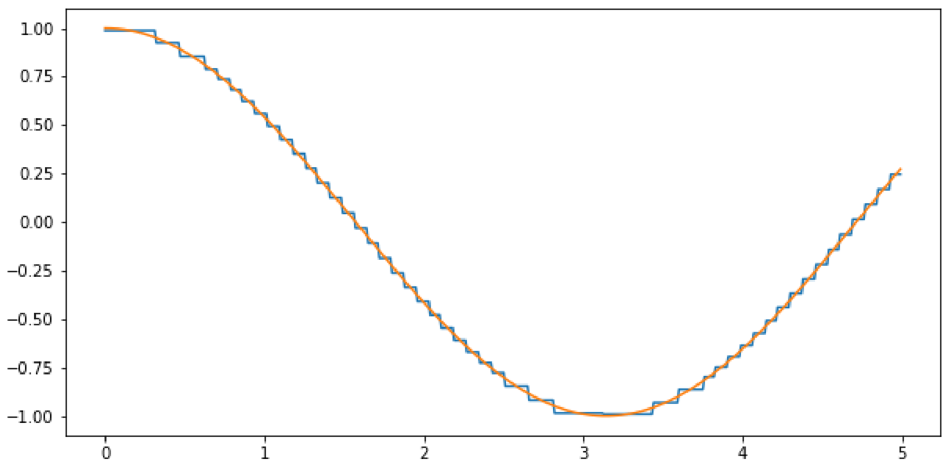

Figure 1.2: Numerical simulation with $f(x) = \cos x$

From this example we remark that the size of the intervals are not the same. Before running the algorithm, we do not know how many calculus we will do : that is a typical example of dynamic applications. And we can imagine that we distribute every interval on one processor.

High performance computing is not about approximation of *cosinus* function on $[0\,;1]$ but it is a nice example to start.

A bigger interval and a smaller c can lead to huge trees[1] that can be paralelized. With such a huge system, there are no possibilities to know the entire state of the system. As a consequence it can be hard to know when we heve our c-approximation of f.

[1]if we see interval as nodes of binary tree

Chapter 2

Strong aspects

2.1 Exisiting solution : HCDA

Huang[Huang 1989] and Mattern proposed independently[1] Credit Distribution Algorithm in order to solve termination detection. The idea is to rely on a controlling agent (CA) to detect termination. This controling agent gives credit to processors. Under certain conditions, processors give back their credit to the controlling agent. Once the controlling agent has received all its credit, the task is over and it is announced by the controlling agent.

When a process delivers a message, it gives a fraction[2] of its credit with the message to the recieving process. It keeps the rest of its credit for itself.

Once the process becomes idle, it returns all its remaining credits to the CA. Here is an illustration of a simple example :

On this example I applied a simple rule in credit repartition.

- Half credits go back to CA, half to the descendant nodes
- Equity in credit repartition in descendant nodes

Mattern take credit of form $X = 2^{-Y}$ which is easier to divide in two and that you can store in the form $Y = -log_2 X$. In that sense Y is the number of time that the initial credit amount has been divided in two. It works well if from one process you can have only two descendant, as in the implementation of 1.2

2.2 Solution prposed : CDA

The solution proposed in the article remains a solution based on credit distribution with a controlling agent hence his name. The point of this new rules on credit management is to face actual problems tackeled by high-performance computing. In actual machine, we do not have infinite amount of storage and credits cannot be divided for eternity. By the way, the Mattern's trick does not solve the problem as credit cannot be added for eternity as well. The idea is to force threshold in carrying credits. The solution proposed in the article is to force credits C_{cur} stored by a process

[1]Huang took the name of the algorithm
[2]following a chosen rule

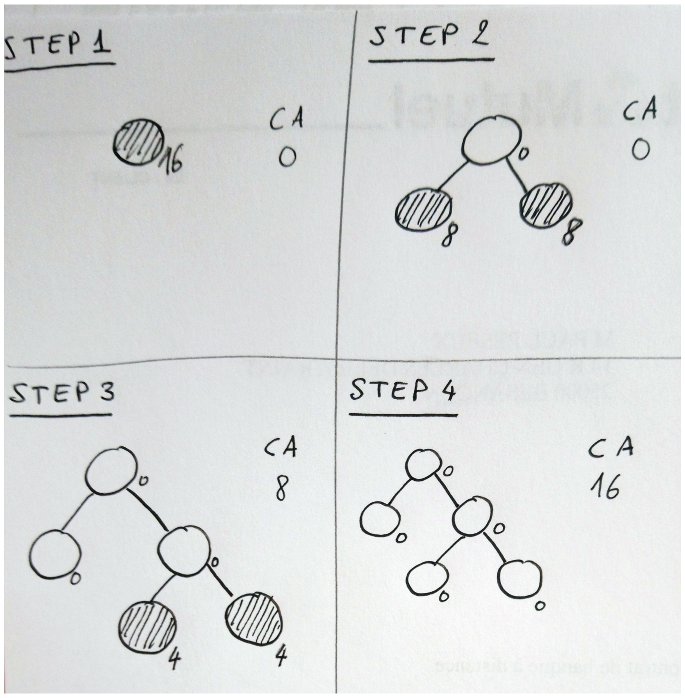

Figure 2.1: HCDA Illustration

to satisfy :

$$C_{\text{con}} \leq C_{\text{cur}}$$

In order to make that feasible, for all application, researchers had to add a novelty : *BORROW* message are in the algorithm itself. If an active process carry less than the chosen threshold, it has to borrow credits to CA. The name *borrow* is well found, as those credits will be returned to CA at the end of the application. As CA gives PC_{init} credits at the beginning, CA detects termination as soon as $C_{returned} = PC_{init} + C_{borrowed}$.

If I understand well, there is an equivalence of CDA and HCDA if

- $C_{\text{con}} = 1$
- $C_{init}(CDA) = C_{init}(HCDA).M_d$

where M_d is the maximal number of division of $C_{init}(HCDA)$

2.3 Token ring

The token ring is the first example given in the article. It is a well known case of study due to its simplicity. At each step there is only one process that is active and with probability q it stops. If it does not stop it continues on a other process chosen randomly.

It

aims to emulate an application where the workload moves across processes during the execution.

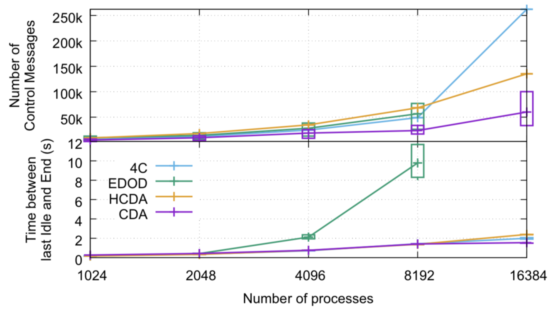

Figure 2.2: Figure taken from the article

It is very interesting to notice that eventhough CDA improves HCDA in order to be better in HPC which is practical application, it also outperforms HCDA, 4C and EDOD in the token ring case in the theoretical case. Indeed if $P \leq \frac{1}{q}$ then

$$\mathbb{E}(CDA) \leq \max\left[\mathbb{E}(HCDA), \mathbb{E}(4C), \mathbb{E}(EDOD)\right]$$

The case where $P \leq \frac{1}{q}$ is not the case in experiments used to draw *Figure 3* as $q_b = 0.1$ and $P = 16,384$

It is the only Theorem in the article, that proves how difficult it is to study termination detection in theoretical case. One strength of the article is to present *expremiments* with different values of P.

2.4 Simulations

On simulations presented in the article, we see how CDA outperforms HCDA.

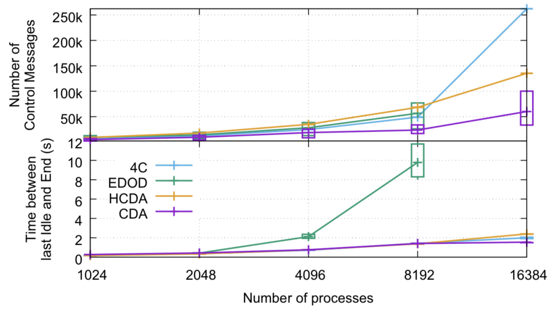

Figure 2.3: Figure taken from the article

It is impressive to notice the gap between those two methods while the difference between the two algorithms is slightly different.

2.5 Implementation

Here is a simple example of the immplementation of CDA :

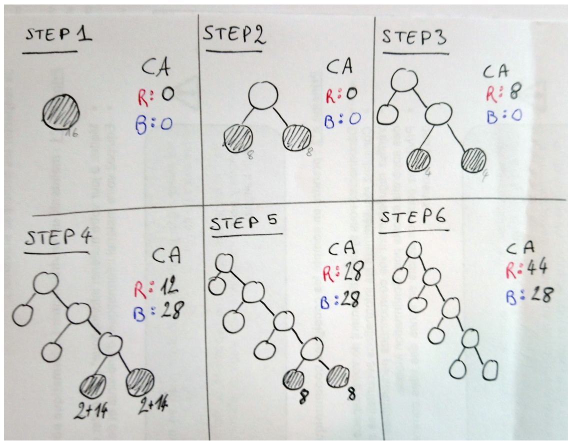

Figure 2.4: CDA Illustration

On this example I used $C_{\text{init}} = 2^4$ and $C_{\text{con}} = 3$. To simplify calculus, processes below C_{con} borrowed credits to $\textit{reach}C_{\text{init}}$, while it should borrow C_{init} credits, no matter what. It is an illustration, but in real case as the number of nodes is huge, there is no possible way to represent trees like that. One has to find efficient way to do it.

This example is part of binary trees, that are studied in the article. It can illustrate the theoretical example given in 1.2.2 where the tree refinement depends on the value of a function on the nodes. It can illustrate a convex function, where you need more and more intervals to approximate it:

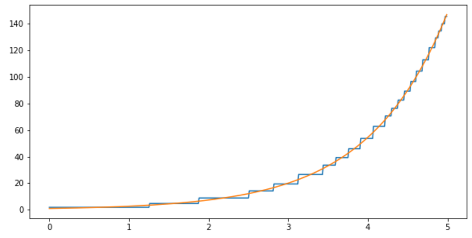

Figure 2.5: Numerical simulation with $f(x) = e^x$

Chapter 3

Weak aspects

3.1 C_{init} and C_{con}

From my point of view, there is a missing discussion in the article. It is clear that if the processes borrows a lot of credits to CA, it will generate a lot of seccondary *BORROW* messages and delay the application. In order to minimize this number, the values of C_{init} and C_{con} are fundamental. Eventhough thoses values are critical, there are not discussed in the article. In the implementation proposed in 2.5, I choosed my own values to make my example relevant. Some curves comparing the impact of C_{con} on the number of generated secondary messages would have been nice.

There is one value $C_{\text{init}} = 2^{32}$ given for the graphs of *Figure 2*, but there are no justification of the value and nothing for C_{con}. As the reason of C_{con} is to prefetch credits, one has to know it will be done.

3.2 BORROW message

There are not mention of *BORROW* message for HCDA [Huang 1989] version. There is something that looks like *BORROW* with

> request for supply

but it is not the same thing and comparing it with the same names in Figure 2 is not clear to me.

3.3 Optimization

Reading the article, I did not clearly understood what has to be optimized. That is the problem tackeled by [Mahapatra 2007] or [Tseng], there are two quantities that can be optimized:

- number of secondary messages
- detection delay

The CDA method want to optimize both, but we do not know the balance between the priorities. The *Figure 3* shows that there are no clear function to optimize, for example $\frac{d}{d_0} + C_m$ where d is the delay, d_0 a constant and C_m the number of control messages.

With such an objective function, it may be easier to find an optimal policy to efficiently detect termination.

Chapter 4

Comparison with related work

4.1 Other algorithms

There exists other way to perform termination detection. In the article, the performances of 4C and EDOD are compared with HCDA and the solution proposed. Those techniques establish state of the art in termination detection.

Conclusion

The presented article proposes a new algorithm based on credit distribution to detect termination in HPC. It improves the former algorithm HCDA by taking into accounts limits of actual systems and by building itself in order to deal with them. This srategy is rewarded in practice by outperforming on experiments. However, there are not theoretical proofs of its superiority on non deterministic binary trees. Moreover the article forget to explain how C_{con} (which is a fundamental parameter) is set up. Working on this parameter looks like promising and could be the subject of an article to come.

Bibliography

[Huang 1989] S. T. Huang. *Detecting termination of distributed computations by external agents.* ICPP, pages 79–84. Pennsylvania State University Press, 1989.

[Mahapatra 2007] Nihar R. Mahapatra et Shantanu Dutt. *An efficient delay-optimal distributed termination detection algorithm.* J. Parallel Distributed Computing, 67(10):1047 – 1066, 2007.

[Tseng] Y. Tseng et R. F. DeMara.