RÉPUBLIQUE FRANÇAISE
*Liberté*
*Égalité*
*Fraternité*

ONERA
THE FRENCH AEROSPACE LAB

# Parallel performance measure and embarrassingly Parallel algorithms

## Performance measure and load balancing

Xavier JUVIGNY, SN2A, DAAA, ONERA
xavier.juvigny@onera.fr

Course Parallel Programming
- January 8th 2023 -

# Table of contents

# Overview

# Speed-up

## Definition

Let's

- $t_s$ : Sequential execution time
- $t_p(n)$ : Execution time on $n$ computing units ;

Speed-up is defined as :

$$S(n) = \frac{t_s}{t_p(n)} \tag{1}$$

## Remark

The sequential algorithm is often different as parallel algorithm. In this case, speed-up measure is not obvious. In particular, following questions must be asked among other questions :

- Is the sequential algorithm optimal in complexity ?
- Is the sequential algorithm well optimized ?
- Is the sequential algorithm exploiting at best the cache memory ?

# Amdahl's law

## Give a limit for the speed-up

- Let's $t_s$ the time running the code in sequential
- Let's $f$ the part (in percent) of $t_s$, relative to a part of the code which can't be parallelized

So

$$S(n) = \frac{t_s}{f.t_s + \frac{(1-f)t_s}{n}} = \frac{n}{1+(n-1)f} \xrightarrow[n\to\infty]{} \frac{1}{f}$$

This law is useful to find the maximal number of computing core to use for an application.

## Limitation of the law

$f$ may change with the volume of input data and bigger input data improve the speed-up.

# Gustafson's law

## Speed-up behaviour with constant volume input data per process

- **Hypotheses** :
  - $t_s \geq 0$ the time to execute the sequential part of the code is indenpendant of the volume of input data ;
  - $t_p > 0$ the time to execute the parallel part of the code is linear relative of the volume of input data.
  - Let's consider $t_s + t_p = 1$ (one unit of time).
- Let's $t_s$ the time token by the execution of the sequential part of the code ;
- Let's $t_p$ the time token by the execution of the parallel part of the code for a fixed amount of data.

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

# Scalability

## Definition

For a parallel program, the behavious of the speed-up when we raises up the number of processes or/and the amount of input data.

## How evaluate the scalability ?

- Evaluate the worst speed-up : For a global fixed amount of data, draw the speed-up curve in function of the number of processes ;
- Evaluate the best speed-up : For a fixed amount of data per process, draw the speed-up curve in function of the number of processes ;
- In concrete use of the program, the speed-up may be between the worst and best scenario.

# Granularity

Ratio between computing intensity and quantity of data exchanged between processes

- Sending and receiving data is prohibitive :
  - Initial cost of a message : each message has an initial cost : set the connection, get the same protocol, and so.This cost is constant.
  - Cost of the data transfer : At last, the cost of the data flow is linear with the number of data to exchange
  - These costs are greater the cost of memory operations in RAM
  - Better to copy some sparse data in a buffer and send the buffer than send data with multiple send and receives
- Try to minimize the number of data exchange between processes
- Greater is the ratio between number of computation instructions and messages to exchange, better will be your speed-up !
- Granularity can be improved with non blocking data exchanges.

# Load balancing

## Definition

**All processes execute a computation section of the code in the same time** ;

- Great penalities in time occure if some parts of the code are far away from load balancing ;
- **Example 1** : A function in the code take $t$ seconds for the half of the processes, and $\frac{t}{2}$ for other processes. The maximal speed-up for this function will be :
$$S(n) = \frac{\frac{n}{2}t + \frac{n}{2}\frac{t}{2}}{t} = \frac{3n}{4}$$
- **Example 2** : A function in the code takes $\frac{t}{2}$ seconds for $n-1$ processes, and $t$ for one process. The maximal speed-up for this function will be :
$$S(n) = \frac{(n-1)\frac{t}{2} + t}{t} = \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

**Remark** : Greater is the time token to execute a bad load balancing function, greater is the penality. Don't worry about load balancing for functions taking very small time to execute

# Overview

# Definition

## Embarrassingly parallel algorithm

- Each data used and computed are independant ;
- No data race in multithread context ;
- No communication between processes in distributed environment

## Property

- In distributed parallel context, no data must be exchanged between processes to compute the results ;
- In shared parallel environment, parallelization is straightforward, but beware to the memory bound computation ;
- In distributed environment, the memory bound limitation is not a trouble ;
- If data are contiguous and algorithm vectorizable, can be ideal on GPGPU for performance.

# First example : Vector addition

Add two real vectors of dimensions $N$

$$w = u + v;\ u, v, w \in \mathbb{R}^3$$

## Ideas

- For load balancing, scatter the vectors in equal parts among the threads or processes
- Each process/thread computes a part of the vector addition
- In distributed memory, the result is scattered among processes !

## Some properties

- Memory access and computing operation have the same complexity : **On shared memory, memory bound limits the performance**
- On distributed memory, each process uses his own physical memory and none data must be exchanged : Speed-up may be linear relative to the number of processes (if data intensity is enough)

# Example : Block diagonal matrices multiplication $C = A.B$ (1)

Matrix-matrix multiplication $C = A.B$ where

$$A = \begin{pmatrix} A_{11} & 0 & \dots & 0 \\ 0 & A_{22} & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & A_{nn} \end{pmatrix}, B = \begin{pmatrix} B_{11} & 0 & \dots & 0 \\ 0 & B_{22} & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & B_{nn} \end{pmatrix}.$$

where $d_i = \dim(A_{ii}) = \dim(B_{ii})$ ($n$ independant matrix-matrix multiplications)

## Problematic

Close to the vector addition multiplication, but :

- Dimensions of diagonal blocks are inhomogeneous & for each diagonal block, computation complexity : $d_i^3$.
- How distribute diagonal blocks among processes to obtain nearly optimal load balancing ?

# Example : Block diagonal matrices multiplication $C = A.B$ (2)

algorithm to distribute diagonal blocks among processes

## Example of algorithm to distribute the diagonal blocks

- Sort diagonal blocks with dimension decreasing ;
- Set "weight" to zero for each process
- Distribute biggest triplet blocks $A_{ii}$, $B_{ii}$, $C_{ii}$ among processes and add each $d_i$ in the weight of each process ;
- While some diagonal blocks are not distributed :
  - Add the biggest block which is not distributed to the process having the smallest weight
  - Add the relative $d_i$ at the weight of the process

**Remark** : All processes compute the distribution of the diagonal blocks. It's better to do same computation for all processes than one computes and sends result to other processes.

# Third example : Syracuse series (1)

## Definition of syracuse serie

$$\begin{cases} u_0 \text{ choosen} \\ u_{n+1} = \begin{cases} \frac{u_n}{2} \text{ if } u_n \text{ is even} \\ 3.u_n + 1 \text{ if } u_n \text{ is odd.} \end{cases} \end{cases}$$

## Property of syracuse serie

- One cycle exists : $1 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow \cdots$
- A conjecture : $\forall u_0 \in \mathbb{N}$, the serie reaches the cycle above in a finite number of iterations

## Some definition

- **Length of flight** : number of iterations for a serie to reach the value 1 ;
- **Height of the flight** : maximal value reached by a serie

**The goal of the program :** compute the length and the height of flight for lot of odd values for $u_0$

# Third example : Syracuse series (2)

## Problematic

- Each process computes the length and the height for a subset of initial values $u_0$ ;
- The computation intensity depends of the length of each syracuse serie ;
- It's impossible to know a priori the computation complexity to compute the height and the length of a serie
- Use a dynamic algorithm on "root" process (the "master" process) to distribute serie among other processes ("slaves")

## Master's Algorithm

- Send a small pack of series to each slave processes ;
- While(some pack of series to send) do
- Wait slave asking series and send next pack ;
- end While
- Send terminaison id to all slave processes ;

## Slave's Algorithm

- While (receive some series to compute in a pack)
- Compute each serie of the pack ;
- end While

**Remark** : A task is composed of a pack of series to have a good granularity.

# Overview

# Nearly embarrasingly parallel algorithm

## Definition

Independant computation for each processes with a final communication to finalize the computation.

## Examples

- Dot product of two vector in $\mathbb{R}^n$ ;
- Compute an integral ;
- Matrix-vector product ;

## Non embarrasingly parallel algorithm examples

- Parallel sort algorithms ;
- Matrix-matrix product ;
- Algorithms based on domain decomposition methods ;

# Integral computation

## Integral computation

- Integral computation based on Gauss quadrature formulae :

$$\int_a^b f(x)dx \approx \sum_{i=1}^{Ng} \omega_i f(g_i)$$

  where $\omega_i \in \mathbb{R}$ the weights and $g_i \in \mathbb{R}$ integrals points.
- In fact, Gauss quadrature are given on $[-1;1]$ interval : some variable modification to do in the integral !;
- $\{g_1 = 0, \omega_1 = 2\}$ : Order 1 Legendre Gauss quadrature ;
- $\left\{\left(g_1 = -\frac{\sqrt{2}}{2}, \omega_1 = \frac{5}{9}\right), \left(g_2 = 0, \omega_2 = \frac{8}{9}\right), \left(g_3 = +\frac{\sqrt{2}}{2}, \omega_3 = \frac{5}{9}\right)\right\}$ : Order 3 Legendre Gauss quadrature
- **Remark** : Order *n* means than quadrature compute the exact value of the integral for polynoms of degree lesser or equal to *n*.
- To compute better approximation of the integral, we subdivide the interval in several smaller intervals

# Parallel integral computation

$$I = \int_a^b f(x)dx = \sum_{i=1}^{N} \int_{a_i}^{b_i} f(x)dx = \sum_{i=1}^{N} I \text{ where } a_1 = a,\ b_N = b \text{ and } a_i < b_i < a_{i+1}$$

## Main ideas

- Scatter sub-intervals among the processes $P$ to compute partial sums :

$$S_p = \sum_{[a_i;b_i]\in P} \int_{a_i}^{b_i} f(x)dx$$

- Use reduce to compute the integral value (global sum) :

$$S = \sum_{p=1}^{nbp} S_p$$

# Matrix-vector product

Let's a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $u \in \mathbb{R}^m$.
The goal of this algorithm is to compute the matrix-vector product :

$$v = A.u \in \mathbb{R}^n \text{ where } v_i = \sum_{j=1}^{m} A_{ij}.u_j$$

Two possibilities to parallize this algorithm :

- Partitionning the matrix per block of rows :
- Partitionning the matrix per block of columns and the vector $u$ per block of same size.

The goal is to split the computation between processes and use a global communication operation to get the final result.

# Matrix-vector product per rows splitting

Let's

$$A = \left( \begin{array}{c} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_I \\ \hline \vdots \\ \hline A_N \end{array} \right) \text{ where } \forall I \in \{1, 2, \ldots, N\}, A_I \in \mathbb{R}^{\frac{n}{N} \times m}.$$

## Algorithm

- Each process computes a part of $v$. The process $I$ computes $V_I = A_I.u \in \mathbb{R}^{\frac{n}{N}}$
- To compute another matrix-vector product with the new vector, we need to gather the vector in all processes (only necessary for distributed parallel algorithm).

# Matrix-vector product per columns splitting

Let's

$$A = (A_1|A_2|\dots|A_I|\dots|A_N) \text{ and } u = \left(\begin{array}{c} U_1 \\ \hline U_2 \\ \hline \vdots \\ \hline U_I \\ \hline \vdots \\ \hline U_N \end{array}\right) \text{ where } \forall I \in \{1,2,\dots,N\}, A_I \in \mathbb{R}^{n \times \frac{m}{N}} \text{ and } U_I \in \mathbb{R}^{\frac{m}{N}}$$

## Algorithm

- Each process computes a partial sum for $v$. Processus $I$ computes

$$V_I = A_I.U_I \in \mathbb{R}^n$$

- Finnaly, a sum reduction is done to get the final result : $v = \sum_{I=1}^{N} V_I$

# Bhudda set

Let's consider the complex recursive mandelbrot serie :

$$\begin{cases} z_0 = 0, \\ z_{n+1} = z_n^2 + c \text{ where } c \in \mathbb{C} \text{ chosen.} \end{cases}$$

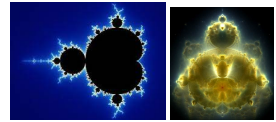

Figure – Mandelbrot (left) and Bhudda's (right) set

## Property

- Serie divergent if $\exists n > 0; |z_n| > 2$ ;
- Region of interest : the disk $\mathcal{D}$ of radius 2 ;
- In some region of the disk, possible to prove the converger
- But chaotic convergence behaviour in some region of $\mathcal{D}$ !

## Mandelbrot and bhudda sets

- Mandelbrot's set :
  color $c$ with "divergence speed" of relative series
- Bhudda's set :
  Color orbit of divergent series

# Bhudda's set algorithm

## Algorithm

- Draw $N$ random values of $c$ in the disk $\mathcal{D}$ where the relative series diverge ;
- Compute the orbit of this serie until divergence and increment the intensity of the pixel representing each value of the orbit ;

## Parallelization of the algorithm

- Master-slave algorithm to ensure load balancing ;
- For granularity, one define a task as a pack of random values $c$ ;