

Parallel performance measure and embarrassingly Parallel algorithms

Performance measure and load balancing

Xavier JUVIGNY, SN2A, DAAA, ONERA

xavier.juvigny@onera.fr

Course Parallel Programming

¹ONERA, ²DAAA
- January 22th 2024 -

Table of contents

① Performance tools

② Embarrassingly parallel algorithms

③ Nearly embarrassingly parallel algorithm

Overview

1 Performance tools

2 Embarrassingly parallel algorithms

3 Nearly embarrassingly parallel algorithm

Speed-up

Definition

Let

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

Speed-up is defined as :

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

Remark

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

- Is the sequential algorithm optimal in terms of complexity ?
- Is the sequential algorithm well-optimized ?
- Is the sequential algorithm exploiting the cache memory to the best extent ?

Amdahl's law

Give a limit for the speed-up

- Let t_s be the time necessary to run the code sequentially ;
- Let f be the fraction of t_s relative to the part of the code that **cannot be parallelized**.

So, the best expected speedup is :

$$S(n) = \frac{t_s}{f \cdot t_s + \frac{(1-f)t_s}{n}} = \frac{n}{1 + (n-1)f} \xrightarrow{n \rightarrow \infty} \frac{1}{f}$$

This law is useful for finding a reasonable number of computing cores to use for an application.

Limitation of the law

The value of f may change with the volume of input data, and larger input data may improve the speed-up.

Gustafson's law

The speed-up behavior remains constant with a fixed volume of input data per process

- Let t_s be the time taken by the execution of the sequential part of the code ;
- Let t_p be the time taken by the execution of the parallel part of the code for a fixed amount of data.
- **Hypotheses :**
 - $t_s \geq 0$: The time to execute the sequential part of the code is independent of the volume of input data ;
 - $t_p > 0$: The time to execute the parallel part of the code is linearly dependent on the volume of input data.
 - Let's consider $t_s + t_p = 1$ (one unit of time).

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

Scalability

Definition

For a parallel program, *scalability* is the behavior of the speed-up when we increase the number of processes and/or the amount of input data.

How to evaluate scalability ?

- Evaluate the worst speed-up : **For a globally fixed amount of data**, plot the speed-up curve as a function of the number of processes.
- Evaluate the best speed-up : **For a fixed amount of data per process**, plot the speed-up curve as a function of the number of processes.
- In practical use of the program, the speed-up may fall between the worst and best scenarios.

Granularity

Ratio between computing intensity and the quantity of data exchanged between processes :

- Sending and receiving data can be prohibitive :
 - **Initial cost of a message** : Each message incurs an initial cost, involving setting up the connection, establishing the same protocol, etc. **This cost is constant.**
 - **Cost of the data transfer** : Finally, the cost of the data flow is linear with the number of data to exchange.
 - These costs outweigh the cost of memory operations in RAM.
 - **It's better to copy some sparse data into a buffer and send the buffer, rather than sending scattered data with multiple sends and receives.**
- **Try to minimize the number of data exchanges between processes.**
- The greater the ratio between the number of computation instructions and messages to exchange, the better your speed-up will be !
- **Low speedup can be improved with non-blocking data exchanges..**

Load balancing

Definition

All processes execute a computation section of the code with the same duration. ;

- Speedup is badly impacted if some parts of the code are far from load balancing.
- **Example 1** : A function takes t seconds for half of the processes and $\frac{t}{2}$ for the other processes. The maximal speed-up for this function will be :

$$S(n) = \frac{\frac{n}{2}t + \frac{n}{2}\frac{t}{2}}{t} = \frac{3}{4}n$$

- **Example 2** : A function 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 : The longer it takes to execute a function with poor load balancing, the greater the penalty. Don't worry about load balancing for functions that take a very small amount of time to execute.

Overview

1 Performance tools

2 Embarrassingly parallel algorithms

3 Nearly embarrassingly parallel algorithm

Definition

Embarrassingly parallel algorithm

- Each used and computed data is independent.
- No data race in a multithread context.
- No communication between processes in a distributed environment.

Property

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

First example : Vector addition

Add two real vectors of dimensions N

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

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 its own physical memory and no 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 independent matrix-matrix multiplications)

Problematic

Close to the vector addition multiplication, but :

- Dimensions d_i of diagonal blocks are inhomogeneous, and for each diagonal block, computation complexity is d_i^3 .
- How to 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 an algorithm to distribute the diagonal blocks

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

Remark : All processes compute the distribution of the diagonal blocks. It is better to do the same computation for all processes than having process 0 compute the distribution and send it to other processes.

Third example : Syracuse series (1)

Definition of Syracuse series

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

Property of Syracuse series

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

Some definitions

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

The goal of the program : compute the length and the height of the flight for a lot of (odd) values of 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 on the length of each Syracuse series ;
- It's impossible to know the computation complexity of a series prior to computing it.
- The problem is not naturally well-balanced ;

⇒ Use a dynamic algorithm on the "root" process (the "master" process) to distribute series among other processes ("slaves")

Master's Algorithm

- Send a small pack of series to each slave process ;
- While (some pack of series to send) do
 - Wait for slaves asking for series and send the next pack ;
- end While
- Send a termination order to all slave processes ;

Slave's Algorithm

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

Overview

1 Performance tools

2 Embarrassingly parallel algorithms

3 Nearly embarrassingly parallel algorithm

Nearly embarrassingly parallel algorithm

Definition

Independent computation for each process with a final communication to finalize the computation.

Examples

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

Non embarrassingly 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 $\omega_i \in \mathbb{R}$ are the weights and $g_i \in \mathbb{R}$ the integration points.

- In fact, Gauss quadrature is 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 that the quadrature computes the exact value of the integral for polynomials of degree less or equal to n .
- To compute a better approximation of the integral, we subdivide the interval into several smaller intervals.

Parallel integral computation

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

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 $A \in \mathbb{R}^{n \times m}$ be a matrix and $u \in \mathbb{R}^m$ a vector.

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 parallelize this algorithm :

- Partitioning the matrix by block of rows :
- Partitioning the matrix by block of columns and the vector u by 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 by rows splitting

Let

$$A = \begin{pmatrix} \frac{A_1}{\hline} \\ \frac{A_2}{\hline} \\ \vdots \\ \frac{A_I}{\hline} \\ \vdots \\ \frac{A_N}{\hline} \end{pmatrix} \text{ where } \forall I \in \{1, 2, \dots, N\}, A_I \in \mathbb{R}^{\frac{n}{N} \times m}.$$

Algorithm

- Each process has some rows of A and all of u
- Each process computes a part of v : the process I computes $V_I = A_I \cdot u \in \mathbb{R}^{\frac{n}{N}}$
- 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 by columns splitting

Let

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

Algorithm

- Each process has some columns of A and some rows of u
- Each process computes a partial sum for v . Process I computes

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

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

Buddha set

Let's consider the complex recursive Mandelbrot series :

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

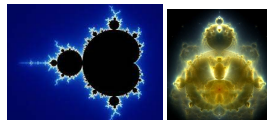


Figure – Mandelbrot (left) and Buddha (right) set

Property

- Series is 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 convergence ;
- But chaotic convergence behaviour in some region of \mathcal{D} !

Mandelbrot and Buddha sets

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

Buddha'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 series 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, define a task as a pack of random values c ;