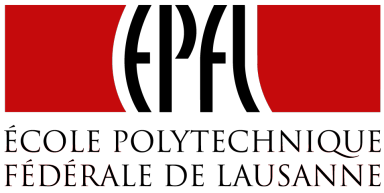


Ecole Polytechnique Fédérale de Lausanne

***Program Parallelization on PC Clusters project*:**

Ant Colony Traveling Salesman Problem Optimization

with MPI

Marc Schär

January 10, 2016

Contents

1 Introduction

In the context of the *Program Parallelization on PC Clusters* course, we have to analyze and implement a parallel application to observe the improvements brought by the parallelization of a serial application. The goal of this project is to anticipate the performance of the parallelized application and to confirm or invalidate the claimed theory.

For this project, I chose to implement an *ant colony optimization algorithm* which gives a solution to the *Traveling Salesman Problem*.

In this report, I will first present a description of the algorithm with a generic pseudo-code, my choices to implement it, the possible problems and solutions and an example of representation of the map. Then, I will make a theoretical analysis of the serial and parallel implementations of the algorithm. Afterwards, I will present results obtained with a MPI implementation, including speedups and comments on the results. Finally, I will make a small conclusion.

1.1 Algorithm

1.1.1 Description

We want to solve the *Traveling salesman problem* with ants. The idea is to have several ants that are spread randomly on the map. Each ant will go through the map and find a path between all cities without visiting twice the same city. At each step, the ant will select the next city with some probability depending on distance and pheromon value. When all ants are done, the pheromons on all the map are reduced by a factor such that edges that are less used have a smaller value. Finally, it updates the visited edges of the best path[1] with pheromons to give more importance to this path for the next iterations. Then, it starts from the beginning until a termination condition is met.

The final goal is to find the shortest path that starts in a city and that arrives in this city going only once through each other city.

The standard termination condition is defined as a maximum number of iterations and/or the same shortest solution after a certain number of iterations.

1.1.2 Pseudo-code

The general algorithm of the *ant colony optimization algorithm* is the following[2] :

```
while (termination condition) {
  for each ant :
    has to visit all cities once
    chooses next city based on distance and pheromon value
  endfor

  reduce the global pheromon values
  update pheromons based on the best path
}
```

[1] It is a choice to keep only the best path between all ants.

[2] This algorithm is based on the *Ant Colony optimization algorithms* found on wikipedia (in french) : https://fr.wikipedia.org/wiki/Algorithme_de_colonies_de_fourmis

1.1.3 Possible problems and solutions

There are several problems that can happen with this algorithm :

- The ant can be stucked in a city if the graph is not fully connected.
  - A possible solution is to kill ants that are in this situation;
  - Another possibility is to transform the graph in a fully connected one with unreasonnable edges for non existing paths[3].
- A suboptimal solution can be found
  - To solve this, it is sufficient to run the algorithm several times[4].
- An isolated city with only one connection to the rest of the graph will make the resolution of the *TSP* impossible.
  - The solution is to assume that each city has at least two connected edges.
- To parallelize this problem, the exchange of pheromon values can be a bottleneck (lot of communications).
  - A possible solution is to run several iterations locally before sharing the result;
  - Another possibility is to send only the updated values and not all the matrix.
  - The exchange of informations can be done using broadcasting between all nodes instead of giving all results to the master that will redestribute them to each node.

1.1.4 Edge selection

An ant will select a path according to some probabilities that includes edge's length and pheromon values :

$$p_{ij}^k(t) = \begin{cases} \frac{\tau_{ij}(t)^\alpha \cdot \eta_{ij}^\beta}{\sum_{l \in J_i^k} \tau_{il}(t)^\alpha \cdot \eta_{il}^\beta} & if \quad j \in J_i^k \\ 0 & if \quad j \notin J_i^k \end{cases}$$

where J_i^k is the list of possible moves for an ant k in the city i, η_{ij} is the *visibility*, which is the inverse of the distance between two cities i and j and τ_{ij} the intensity of a path (the pheromon values). α and β are parameters to control the algorithm.

Finally, I also define $\rho\tau_{ij}(t)$ which represents the pheromon evaporation that is computed at each iteration.

1.1.5 Pheromon update

At the end of each *external* loop, the pheromon values are updated according to the best path of the current iteration. To spread pheromons such that shortest paths between several iterations will have more pheromons than longer ones, I define formula presented in figure 1.

[3] I chose this solution.

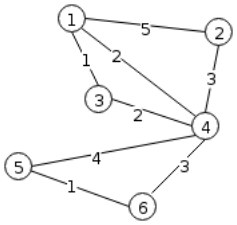
[4] With some improvements for the parallel implementation, the optimal solution will possibly not be reached often.

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{Q}{L^k(t)} & if \quad (i,j) \in T^k(t) \\ 0 & if \quad (i,j) \notin T^k(t) \end{cases}$$

Figure 1: *Update pheromons definition. $T^k(t)$ is the best path of ant k at iteration t, $L^k(t)$ is the length of the path and Qi (I chose $Q = 1$) is a parameter to be defined.*

1.1.6 Representation

To represent this algorithm in a computer, we have to model a map with a graph. Each vertex will represent a city and edges will represent connections between cities. Each edge has a weight representing the distance between all cities. As we want to have a fully connected graph, edges that connect unlinked cities will have a maximal weight (which will be defined in the implementation of the algorithm). Figures 2a and 2b show how to represent the map.



(a) *A map composed by cities and paths*

$$\begin{pmatrix} 0 & 5 & 1 & 2 & \infty & \infty \\ 5 & 0 & \infty & 3 & \infty & \infty \\ 1 & \infty & 0 & 2 & \infty & \infty \\ 2 & 3 & 2 & 0 & 4 & 3 \\ \infty & \infty & \infty & 4 & 0 & 1 \\ \infty & \infty & \infty & 3 & 1 & 0 \end{pmatrix}$$

(b) *The matrix representing figure 2a.*

Figure 2: *Representation of a map in a computer.*

Each ant will simply be represented by finding a path on the map from different starting points. To find a path, the solution is to select a path from the current city (select an element in a column) being careful to not select an already visited city and selecting a city following the formula in section 1.1.4.

We can note that the pheromon values need to be stored in a matrix, too. This matrix will have the same construction as the one in figure 2b.

2 Theoretical Analysis

2.1 Serial application

2.1.1 Computational complexity

The serial *ant colony optimization algorithm* is composed by a first *initialization* part that initializes all variables, including the matrix that represents the graph and the one with pheromon values (complexity of $O(2 \cdot c^2)$ for c cities). Then, the algorithm will loop until a termination condition is reached[5]. This "big" loop will make at most L iterations. In this loop, there is another loop for simulating ants (we have A ants). In it, there is a loop to go through each city (c cities) where we have to compute the probabilities to go to a specific city ($O(c)$). Finally, we have to perform the pheromon's evaporation ($O(c^2)$) and we have to update the pheromon's matrix with the new pheromon's values from ants ($O(c)$ because we have to update only the best path).

The skeleton of the serial application is like this pseudo-code:

```
init() // 2*c*c operations
while (termination condition) { // at most L loops
  for each ants { // A iterations
    for each city { // c iterations
      find next city to visit // O(1)
    }
  }
  pheromon evaporation // O(c*c)
  update bestPath's pheromons // O(c)
}
```

It gives us a computational complexity of $O(c^2 + L \cdot (cA + c^2 + c))$.

We can define the number of ants to the number of city (even if it is useless to have so much ants) and we can consider that the number of *external* loops is a constant, we have a complexity of $O(L \cdot c^2)$.

With this complexity, we can observe that number of ants and the size of the map are the two factors that makes the computations complex or not.

2.2 Parallel Application

For the parallel implementation of the *ant colony optimization algorithm*, we have to define what is parallelized and what are the content of communications.

2.2.1 Basic Parallel Application

For the analysis below, let us consider we have c cities, A ants, n nodes (including the master) and we assume that the loop has a fixed number of iterations L.

[5] Either the maximum number of iterations is reached or the solution is stabilized.

2.2.1.1 Computation to communication time For the initialization part, the graph will be spread by the master to each node ($O(c^2)$) using broadcasting. And, at the end of each iteration, each node will have the share its result (best path or whole pheromons' matrix) to others using broadcasting ($O(n * c)$ for sharing the best path or $O(n * c^2)$ for sharing the whole matrix).

Then, each node will compute the paths of multiple ants ($O(c \cdot \frac{A}{n})$). We assume that each node has the same number of ants. The computation on each node is exactly the same as in the serial application except that each node has only a subset of ants. We can add that if we want to do several iterations before merging the results, we will simply multiply this cost by a constant, which will not change the bounded complexity.

Then, when they all have finished to compute the new map with new pheromon's values, all nodes send their local best path to the others. The cost of sending all vectors is $O(cn)$.

Finally, each node will merge all received vectors ($O(c \cdot n)$ additions) and will perform the *pheromons evaporation* ($O(c^2)$) and decide if another iteration is needed or not.

Thus, for computation time we have a cost of $O(L \cdot (c \cdot \frac{A}{n} + c \cdot n + c^2))$ and for communication time we have a cost of $O(c^2 + L \cdot c \cdot n)$[6] because each node has to send its best path at each *external* iteration.

The *Computation/Communication* ratio is $\frac{O(L \cdot (c \cdot \frac{A}{n} + cn + c^2))}{O(c^2 + L \cdot c \cdot n)}$ which tells us that the size of the problem will not make the communications less important. In fact, the only way to have more computations than communications is to have $\frac{A}{n} > c$.

We conclude that the communications are really costly for this algorithm.

2.2.1.2 Amdhal's law The Amdhal's law gives us a theoretical speedup when we improve resource for a problem of a fixed size. For the *ant colony optimization problem*, we can define this theoretical upper bound limitation as follows, where n is the number of nodes and f is a fraction of the code that cannot be parallelized :

$$S_n \leq \frac{n}{1 + (n-1) \cdot f}$$

2.2.1.2.1 Without data transfer If we do not account for data transfers, the fraction of the code that cannot be parallelized is when the master creates the matrix at the beginning of the program and when all nodes share their best paths.

Considering the complexities defined above, the serial part will have a complexity of $O(c^2)$ to initialize the matrix and then a complexity of $O(L \cdot c \cdot n)$ to update the matrix. It gives us $c \cdot (L \cdot n + c)$ operations.

For the parallel part, we have $L \cdot c \cdot \frac{A}{n} \cdot c + L \cdot c^2$ operations[7].

Thus,

$$f = \frac{c \cdot (L \cdot n + c)}{c \cdot (L \cdot n + c) + L \cdot c^2 \cdot (1 + \frac{A}{n})} = \frac{L \cdot n + c}{L \cdot n + c + L \cdot c \cdot (1 + \frac{A}{n})}$$

Here, we observe that A and n are important. To have a small f, we need to have more ants than nodes. It is logical, because to have an efficient parellelization of this program, nodes must have something to compute (the computation time on nodes must be big compared to communications.

Table 3 and figure 4 show us the important relation between the number of ants and nodes. They also show us that more we have nodes, more we have serial code execution.

[6] We share only the best path from each node.

[7] We go through each city and in each city we have to compute the probabilities for each other cities and finally we update the peromons' matrix.

Nodes \ Ants	16
1	0.000070583252947%
2	0.000244384705961%
4	0.000839294992207%
8	0.002725882587594%
16	0.008034917170916%

Figure 3: *Percentages of serial code execution without considering data transfer for 16 ants, 5000 iterations and 1000 cities.*

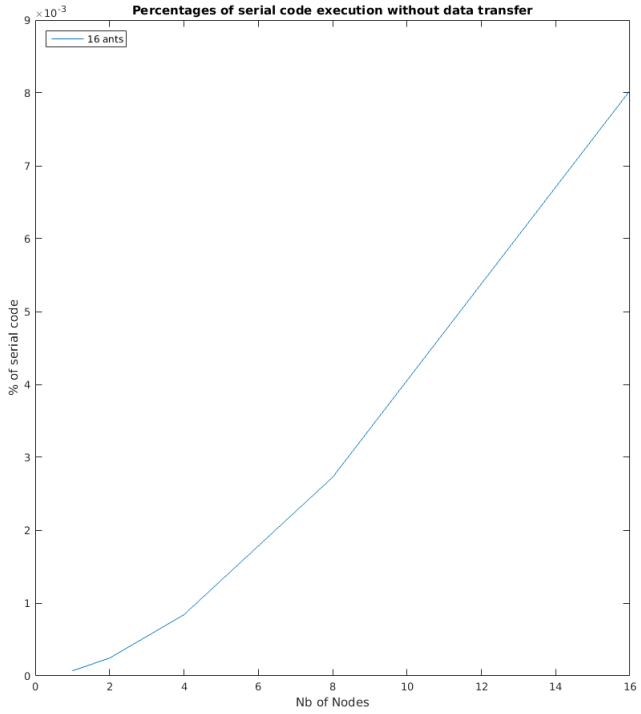


Figure 4: *Percentages of serial code execution without considering data transfer for 16 ants, 5000 iterations and 1000 cities.*

2.2.1.2.2 More accurate estimation Now, if we consider the transfer of data between nodes, we must modify the value of f. Indeed, we have to consider the communication time in our computations. To send a matrix from the master to each node, we have c^2 values sent to n nodes using broadcast and we have $c \cdot n$ values returned from n nodes L times. Thus, we have :

$$f = \frac{c \cdot (L \cdot n + c) + c \cdot (c + L \cdot n)}{c \cdot (L \cdot n + c) + c^2 + c \cdot L \cdot n + L \cdot c^2 \cdot (1 + \frac{A}{n})} = \frac{2 * (c^2 + c \cdot L \cdot n)}{2 * (c^2 + c \cdot L \cdot n) + L \cdot c^2 \cdot (1 + \frac{A}{n})}$$

Again here, we observe (figure 6 and table 5) that the number of ants compared to the number of nodes is important to have a good speedup. We can already affirm that adding ressources for solving this problem will not be efficient if the problem is not big enough (not enough cities and ants).

Nodes \ Ants	16
1	0.000070583252947%
1	0.000244384705961%
1	0.000839294992207%
1	0.002725882587594%
1	0.008034917170916%

Figure 5: *Percentages of serial code execution with considering data transfer for 16 ants, 5000 iterations and 1000 cities.*

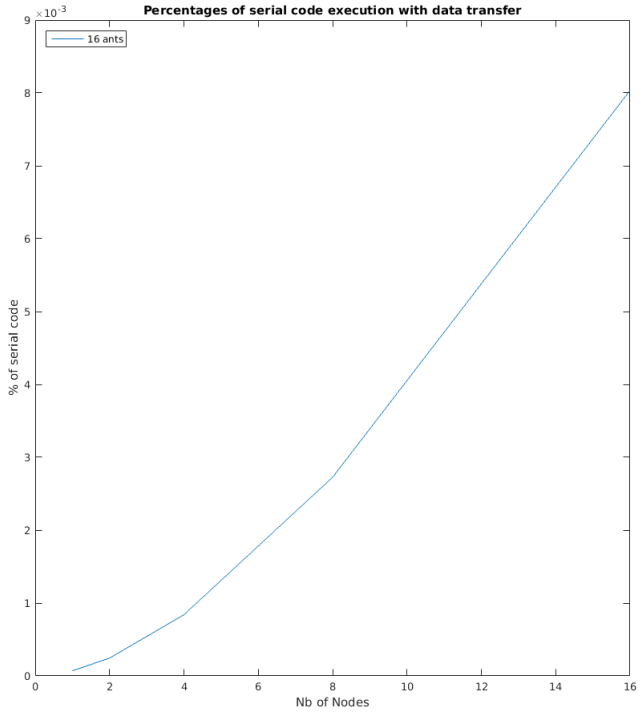


Figure 6: *Percentages of serial code execution with considering data transfer for 16 ants, 5000 iterations and 1000 cities.*

We can observe that the more accurate estimation has exactly the same result as the one without considering data transfer. We can explain that with the fact that the code for communications has te same importance in this analysis as the initialization of the map and the pheromons update. Thus, the fraction of serial code stays the same (even if in reality, the communication will take much longer than updating two matrices).

2.2.2 Improved Parallel Application

As we can observe with the parallel implementation where we send the whole matrix at each iteration, we can reduce the communications with computing several iterations locally before sending the matrix

to the master (we reduce the communications having more accurate local solutions before merging). We also can send to the master only the best path over several iterations.

As we want to improve the performances of the parallel application we can consider both improvements together. Thus, we need to define the variable l which represents the number of iterations to perform locally before merging the best path to the master. The theoretical speedup upper bound is presented below. We will do here the prediction for these improvements, but we need to be sure that the solutions provided with these improvements is still good compared to the serial implementation. This will be done in the last part of this report.

2.2.2.1 Computation to communication time

The initialization part stays the same as in the basic parallel application : $O(c^2)$ to initialize the matrix and $O(c^2)$ to send it to the nodes.

For the computation complexity on nodes, we have now a complexity of $O(L' \cdot (l \cdot c \cdot \frac{A}{n}))$. The update of pheromon values is done locally and is kept between local and external iterations[8].

Then, for the merge with other nodes, we only send the best path (which needs to be kept in a local variable and updated after each ant execution). Thus the complexity is $O(L' \cdot c \cdot n)$.

Finally, every node has to merge all other nodes results : $O(L' \cdot c \cdot n)$.

The *Computation/Communication* ratio is

$$\frac{c^2 + L' \cdot (l \cdot c \cdot \frac{A}{n} + c \cdot n)}{c^2 + L' \cdot c \cdot n} = \frac{c + L' \cdot (l \cdot \frac{A}{n} + n)}{c + L' \cdot n}$$

Here we can conclude that the larger is the problem (in term of ants, city and thus local iterations), the smaller is the impact of communication. Nevertheless, more we have nodes, more are the communications important (this seems totally logic).

2.2.2.2 Amdahl's law

2.2.2.2.1 Without data transfer

Without considering the data transfer, we have to compute in a serial manner the initialization of the matrices (c^2) and the merge of all best paths ($L' \cdot n \cdot c$). For the parallel part, we have the computation of the best path for each ant with several local iterations ($L' \cdot (l \cdot (c^2 \cdot \frac{A}{n}))$).

Thus, we have the following non-parallel fragment :

$$f = \frac{c^2 + L' \cdot n \cdot c}{c^2 + L' \cdot n \cdot c + \frac{L' \cdot l \cdot c^2 \cdot A}{n}} = \frac{c + L' \cdot n}{c + L' \cdot n + \frac{L' \cdot l \cdot c \cdot A}{n}}$$

We observe (table 7 and figure 8) again that the number of ants and the number of local iterations is important to maximize the speedup. Thus, more we have nodes, more we need to have a bigger problem to solve to keep a reasonnable speedup.

[8] Thus, we have some history in each node to help it to find a solution faster.

Nodes \ Ants	16
1	0.012941008998707%
2	0.026665955574518%
4	0.055996864175606%
8	0.119985601727793%
16	0.259932417571431%

Figure 7: *Percentages of serial code execution without considering data transfer for 16 ants, 100 external iterations, 50 local iterations and 1000 cities. Results must be multiplied by* 10^{-3}

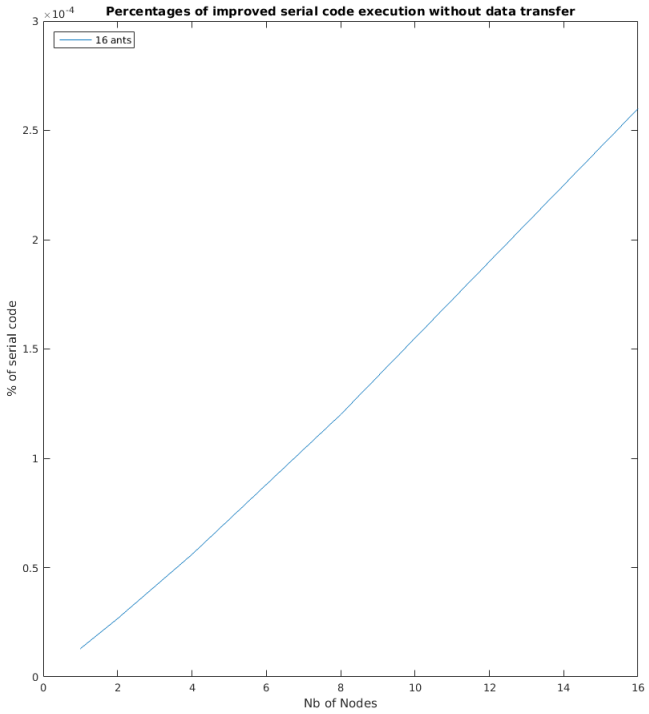


Figure 8: *Percentages of serial code execution without considering data transfer for 16 ants, 100 external iterations, 50 local iterations and 1000 cities.*

2.2.2.2.2 More accurate estimation For having a more accurate estimation, we consider the communication time. As we defined above, we consider the send of the whole matrix (c^2) to each node and the send of all best paths ($L' \cdot c \cdot n$). Thus, we have the formula presented in figure 9.

$$f = \frac{2 * (c^2 + L' \cdot n \cdot c)}{2 * (c^2 + L' \cdot n \cdot c) + \frac{L' \cdot l \cdot c^2 \cdot A}{n}} = \frac{2 * (c + L' \cdot n)}{2 * (c + L' \cdot n) + \frac{L' \cdot l \cdot c \cdot A}{n}}$$

Figure 9: *Amdhal's law for improved implementation.*

Again, we can observe (table 10 and figure 11) that the number of ants and the number of local

iterations is important to define the speedup, but, here, the number of cities makes the communication more dominant. It is logic, because the vectors will be bigger so the communications increase with the number of cities and the number of nodes.

Nodes \ Ants	**16**
1	0.000027499243771%
2	0.000059996400216%
4	0.000139980402744%
8	0.000359870446639%
16	0.001038919523695%

Figure 10: *Percentages of serial code execution with considering data transfer for 16 ants, 100 external iterations, 50 local iterations and 1000 cities.*

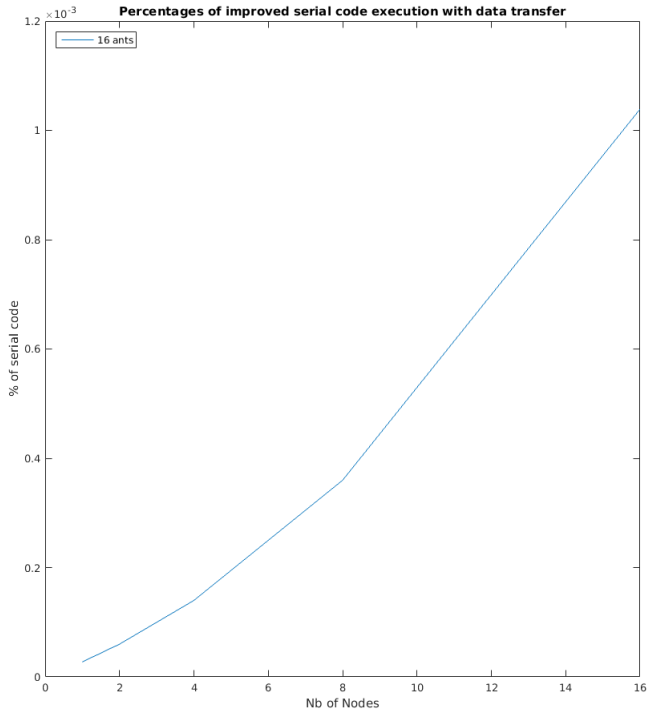


Figure 11: *Percentages of serial code execution with considering data transfer for 16 ants, 100 external iterations, 50 local iterations and 1000 cities.*

2.2.3 Conclusion on Amdahl's law analysis

The vision of Amdhal's law is pessimist because the communications will take a lot of time if we increase the resources without increasing the problem. On the contrary, Gustafson's law is more optimistic because it adapts the task to the number of computation resources. Nevertheless, Amdahl's law gives us a good idea of how many cores we need for a specific problem's size and in the *ant colony optimization*

problem it is important to avoid having too much nodes compared to the size of problem, according to the theoretical analysis.

2.3 Theoretical analysis with MPI

Now, I will establish an analysis of the speedup using the framework *Message Passing Interface (MPI)* which allows several computational nodes to work together in a distributed memory fashion.

As discussed in part 2.2, we have the following sequence of message passing for one master and 3 nodes[9].

For spreading the matrix into all nodes, we can use `MPI_BCAST` which sends to all nodes the same data. It allow us to improve a bit the communications.

2.3.1 Sequence of message passing

We have a kind of *DPS flow graph* like in figure 12 where we can observe the general scheme of the algorithm. We also have two *MPI* operations which are when the master send the matrix to each node and when they send back their own best path.

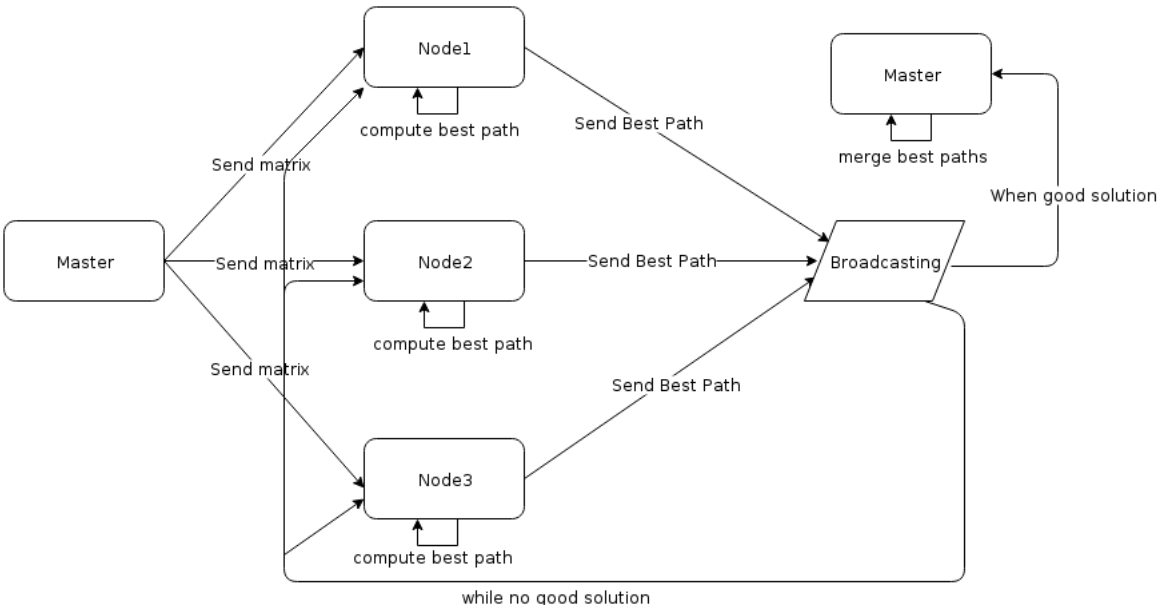


Figure 12: *General scheme of the application. The broadcast allows nodes to share their best paths together.*

The sequence of message passing is presented in figure 13 for one iteration, without considering the local computations. The timeline has no meaning in term of size. This sequence diagram is here only to show in which order are sent messages between nodes and master.

2.3.2 Approximative Timing Diagram

Figure 14 presents an approximative timing diagram of computations and communications between one master and 2 slave nodes. We suppose, for keeping small this diagram, that there are only 3 cities and

[9]Of course, in the real implementation, the master will work as a node, too.

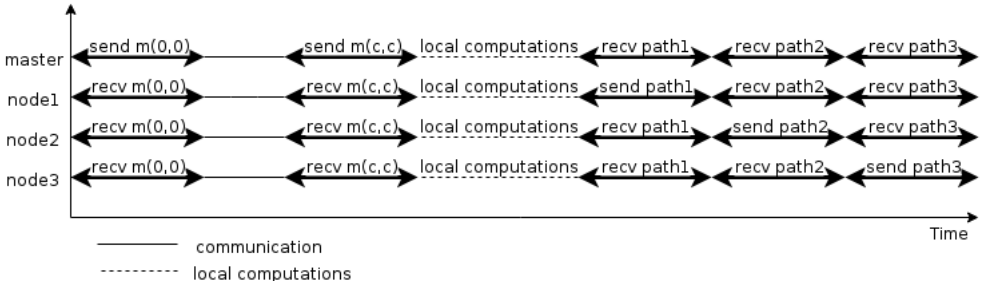


Figure 13: *Sequence of message passing*

3 ants (one per node). The computation phases are made bigger than the communication ones, but in reality, with this example, the communications should take more time[10]. I also consider that all similar operations take the same time to be executed.

In fact, this is true. Indeed, all comunications are sending only one value so, without considering problems on the network, it should always take the same time to be executed. And even if we send the whole matrix at one time, the matrix will always have the same size, and we can expect to always have the same communication time. For computations, if all nodes have the same number of ants to manage, they will have a similar computation time (if the machines have an identical architecture and hardware) because in all cases the map has the same size, and each city has the same number of neighbours to consider.

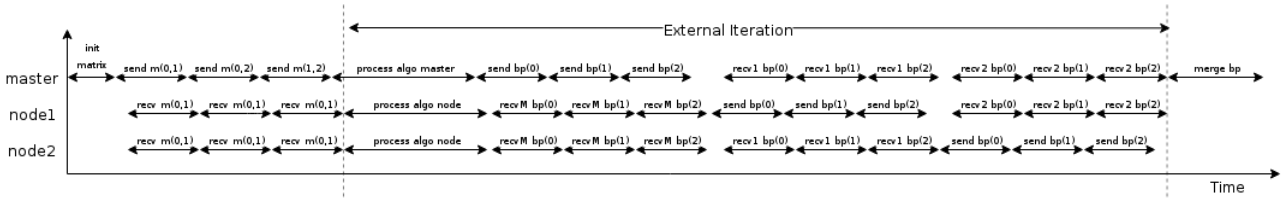


Figure 14: *Timing diagram for initialization + one iteration with 3 nodes. bp means best path.*

To have a better idea of the different timings, we can suppose we have a network with a throughput of 125 MB/s[11] for data. Knowing that each message sent by a node or the master will contain n integers (the indices of cities of the best path) and that an integer is 4 bytes, we can assume that a message takes $3.0518 \cdot 10^{-8}$ seconds to be sent[12].

For the computation time, I ran one iteration of the algorithm with one ant on an *Intel core i5* machine with 4 CPU at 2.60 GHz for a map of 1000 cities. It took 0.021076 seconds. For initializing variable (especially map and pheromons matrix), it took 0.139912 seconds, even if this time can vary because the map is read from a file. For the merging part, it took 0.000009 seconds.

With these values, we can do some computations to estimate the time for computations with multiple nodes and multiple ants for a fixed map size of 1000 cities[13]. It is presented in figure 15[14].

We can observe in figures 15 and 16 that adding nodes improves the performances of the program. Nevertheless, we can already expect that with the variance of the network speed, the practical speedup will not be as good as here.

[10] I consider a huge problem where the communications are less important than computations.
[11] 125MB = 131'072'000 bytes.
[12] We neglect the small latency.
[13] Of course, I will try with different map sizes when the parallel algorithm will be implemented.
[14] The communication parts are not considered when there is only one node.

nodes \ ants	16
1	1686.230113000000
2	843.199313000000
4	421.697713000000
8	210.974513000000
16	105.668113000000

Figure 15: *Theoretical computations for 1000 cities and 5000 iterations (100 external iterations and 50 internal ones) based on one machine measurements. The values are computed as follows (in seconds) :* $t_{init} + 1000^2 * t_{send} + 100 * (50 * (\lceil \frac{n_{ants}}{n_{nodes}} \rceil \cdot t_{computation}) + n_{nodes} \cdot 1000 \cdot t_{send} + t_{merge}) + t_{merge})$.

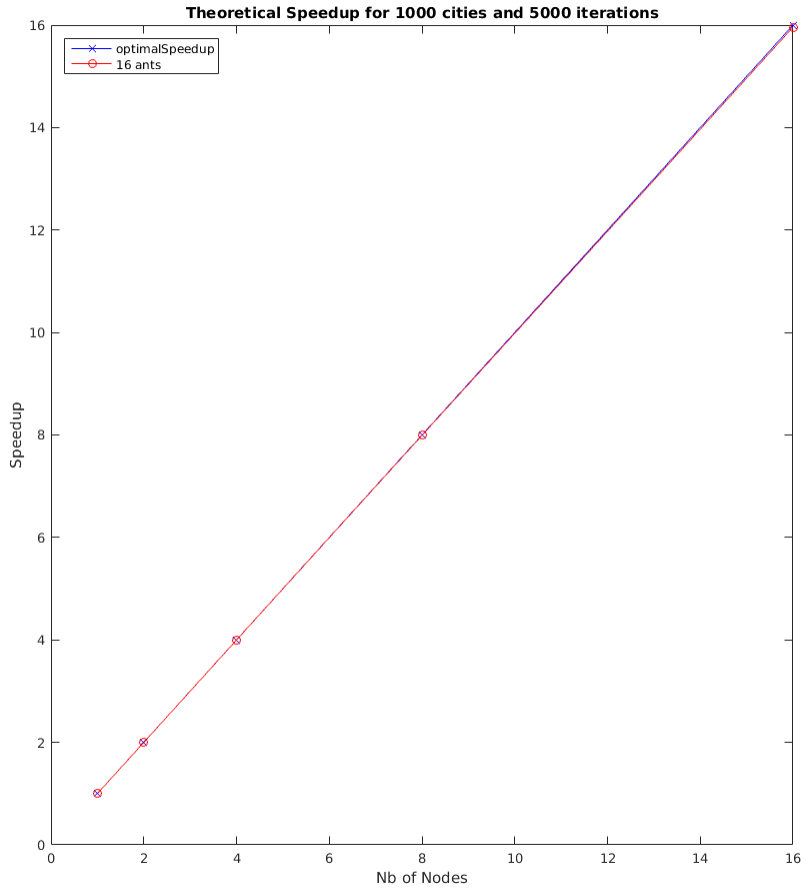


Figure 16: *Theoretical speedup for 1000 cities and 3000 iterations (30 * 100) based on figure 15.*

2.3.3 Critical Path and Theoretical Speedup

As we can observe on figure 14, the critical path is the following one :

$init\ matrix \rightarrow send\ m(0,1) \rightarrow send\ m(0,2) \rightarrow send\ m(1,2) \rightarrow process\ algo\ node \rightarrow recv1\ bp(0) \rightarrow recv1\ bp(1) \rightarrow$

$$recv1\ bp(2) \rightarrow recv2\ bp(0) \rightarrow recv2\ bp(1) \rightarrow recv2\ bp(2) \rightarrow merge\ bp$$

With this critical path, we can define a theoretical speedup. The speedup is defined as follows :

$$S_n = \frac{t_s}{t_p}$$

where t_s is the time of the serial application and t_p is the time of the parallel application.

In this theoretical speedup analysis, we will keep t_s as such[15], but we will define morce precisely t_p. With the critical path, we can define t_p as follows :

$$t_p = t_{init} + t_l + c^2 \cdot t_{send} + L \cdot (t_s \cdot l + t_l + n \cdot c \cdot t_{send})$$

where t_{init} is the time to initialize the matrix, t_l is the minimal time to start a transmission, c is the number of cities[16], t_{send} is the time to send one element of the matrix, L is the number of *external* iterations in the serial program, l is the number of iterations to process locally in a node before sharing with other nodes and n is the number of nodes.

We can reduce t_p :

$$t_p = t_{init} + (L+1) \cdot t_l + t_{send} \cdot (c^2 + n \cdot c \cdot L) + L \cdot l \cdot t_s(n)$$

We also can define t_s as :

$$t_s(x) = \frac{A}{x} \cdot t_{computation}$$

where x is the number of nodes and A the number of ants.

Finally, the speedup is as follows :

$$S_n = \frac{t_{init} + L \cdot l \cdot t_s(1)}{t_{init} + (L+1) \cdot t_l + t_{send} \cdot (c^2 + n \cdot c \cdot L) + L \cdot l \cdot t_s(n)}$$

To have good performances, we need to have t_p as small as possible. Ideally, t_{send} should be really small such that the problem can scale easily.

2.3.3.1 Remarks We can note that, in this analysis, we always compare the serial and the parallel implementations when they have exactly the same number of iterations. It allows us to have acceptable results. Nevertheless, we will see in next section that we can have another termination condition which is a certain cost for a path that stays the same for a certain number of iterations (we assume that the result cannot be better in this case). We will observe that some techniques we use to parallelize are not very good in term of result's optimality (even if it as a really good speedup).

[15] We add to it the initialization phase because it is also considered in the parallel computations.
[16] We send only the upper part of the matrix because it is symmetric and the diagonal is null.

3 MPI Results

In this section, I will present the results obtained with a MPI program for the *ant colony optimization problem*. All following results were obtained from the *Bellatrix* cluster from *SCITAS*. Informations about *Bellatrix* can be found on its official website[17].

3.1 Implementation

For both implementations (serial and parallel), I chose to implement a `utils.h` file to share common functions between all implementations. This file contains functions to help to debug the code (like `printMap()`, `copyVector()` to copy a vector into another one or functions to get the current time.

For solving the problem, I implemented several functions that are useful to find a path or to get the cost of a path, but I also have implemented two functions to choose the next city when an ant is computing its path :

```
/**
 * Compute the probability to go in each city from current city
 **/
void computeProbabilities(int currentCity, double* probabilities, int* path, int* map,
  int nCities, double* pheromons, double alpha, double beta) {

  int i;
  double total = 0;
  for (i = 0; i < nCities; i++) {
    if (path[i] != -1 || i == currentCity) {
      probabilities[i] = 0.0;
    } else {
      double p = pow(1.0 / map[getMatrixIndex(currentCity,i,nCities)],alpha) *
        pow(pheromons[getMatrixIndex(currentCity,i,nCities)], beta);
      probabilities[i] = p;
      total += p;
    }
  }

  // If all the probabilities are really small
  // We select one (not randomly to have always the same behavior)
  if (total == 0) {
    i = 0;
    for (i = 0; i < nCities; i++) {
      if (path[i] == -1 && i != currentCity) {
        probabilities[i] = 1.0;
        total++;
      }
    }
  }

  for (i = 0; i < nCities; i++) {
    probabilities[i] = probabilities[i] / total;
  }
```

[17] http://scitas.epfl.ch/hardware/bellatrix-hardware

```
}

/**
 * Given the current city, select the next city to go to (for an ant)
 **/
int computeNextCity(int currentCity, int* path, int* map, int nCities,
  double* pheromons, double alpha, double beta, long random) {

  int i = 0;
  double *probabilities;
  probabilities = (double*) malloc(nCities*sizeof(double));
  computeProbabilities(currentCity, probabilities, path, map, nCities, pheromons,
    alpha, beta);

  int value = (random % 100) + 1;
  int sum = 0;

  for (i = 0; i < nCities; i++) {
    sum += ceilf(probabilities[i] * 100);
    if (sum >= value) {
      free(probabilities);
      return i;
    }
  }
  free(probabilities);
  return -1;
}
```

As you can observe, the formula presented in section 1.1.4 is implemented in `computeProbabilities()`. You can also see that, according to limitations in number precision, sometimes, we can have all probabilities to 0. In this case, I simply choose to give equal probabilities to each cities that were not visited by the ant.

In `utils.h`, I also implement the function to update the pheromons after a local iteration :

```
/**
 * Update the pheromons in the pheromons matrix given the current path
 *
 * The maximum pheromon value for an edge is 1.
 **/
void updatePheromons(double* pheromons, int* path, long cost, int nCities) {
  int i;
  int* orderedCities = (int*) malloc(nCities*sizeof(int));

  for (i = 0; i < nCities; i++) {
    int order = path[i];
    orderedCities[order] = i;
  }

  for (i = 0; i < nCities - 1; i++) {
    pheromons[getMatrixIndex(orderedCities[i],orderedCities[i + 1], nCities)] +=
      1.0/cost;
```

```
    pheromons[getMatrixIndex(orderedCities[i + 1],orderedCities[i], nCities)] +=
      1.0/cost;
    if (pheromons[getMatrixIndex(orderedCities[i],orderedCities[i + 1], nCities)] > 1)
    {
      pheromons[getMatrixIndex(orderedCities[i],orderedCities[i + 1], nCities)] = 1.0;
      pheromons[getMatrixIndex(orderedCities[i + 1],orderedCities[i], nCities)] = 1.0;
    }
  }
  // add last
  pheromons[getMatrixIndex(orderedCities[nCities - 1],orderedCities[0],nCities)] +=
    1.0/cost;
  pheromons[getMatrixIndex(orderedCities[0],orderedCities[nCities - 1], nCities)] +=
    1.0/cost;
  if (pheromons[getMatrixIndex(orderedCities[nCities - 1],orderedCities[0],nCities)] >
    1.0) {

    pheromons[getMatrixIndex(orderedCities[nCities - 1],orderedCities[0],nCities)] =
      1.0;
    pheromons[getMatrixIndex(orderedCities[0],orderedCities[nCities - 1], nCities)] =
      1.0;
  }
}
```

I had to limit the maximum value of pheromons to 1 because in some cases, it can become bigger than 1 and then, if the program runs long enough, it can diverge to infinity.

3.1.1 Serial

For the serial implementation, I simply take the general algorithm of the *ant colony optimization problem* and I implement it in `C` with a bit of `C++`. Here is the implementation of the algorithm with the whole loop (for details, see code). We must be careful with the "random" part of this algorithm. Indeed, to compare implementations together, we need to have the same random numbers to observe if the result has the same quality between all implementations. Thus, I load a file composed by random numbers and then, everytime I need to have a random number, I take the number from the file (in an incrementally manner).

```
// External loop
  while (loop_counter < iterations &&
    terminationCondition < (long) ceilf(iterations * terminationConditionPercentage)) {

    // Loop over each ant
    for (ant_counter = 0; ant_counter < nAnts; ant_counter++) {
      // init currentPath
      // -1 means not visited
      for (i = 0; i < nCities; i++) {
        currentPath[i] = -1;
      }

      // select a random start city for an ant
      long rand = randomNumbers[random_counter];
```

```
      int currentCity = rand % nCities;
      random_counter = (random_counter + 1) % nRandomNumbers;
      // currentPath will contain the order of visited cities
      currentPath[currentCity] = 0;
      for (cities_counter = 1; cities_counter < nCities; cities_counter++) {
        // Find next city
        rand = randomNumbers[random_counter];
        currentCity = computeNextCity(currentCity, currentPath, map, nCities,
          pheromons, alpha, beta, rand);
        random_counter = (random_counter + 1) % nRandomNumbers;


        if (currentCity == -1) {
          printf("There is an error choosing the next city in iteration %d
            for ant %d\n", loop_counter, ant_counter);
          return -1;
        }

        // add next city to plan
        currentPath[currentCity] = cities_counter;
      }

      // update bestCost and bestPath
      long oldCost = bestCost;
      bestCost = computeCost(bestCost, bestPath, currentPath, map, nCities);

      if (oldCost > bestCost) {
        copyVectorInt(currentPath, bestPath, nCities);
      }
    }

    if (bestCost < antsBestCost) {
      antsBestCost = bestCost;
      terminationCondition = 0;
    } else {
      terminationCondition++;
    }

    // Pheromon evaporation
    for (j = 0; j < nCities*nCities; j++) {
      pheromons[j] *= evaporationCoeff;
    }
    // Update pheromons
    updatePheromons(pheromons, bestPath, bestCost, nCities);

    loop_counter++;
  }
```

As you can see, the main part of the code is quite simple and is composed by less than 100 lines of code.

3.1.2 Parallel

For the parallel implementation, I need to implement communication between nodes. Thus, it will have a lot of communications at the beginning of the execution to share all useful informations (map, number of ants for each node for example) between all nodes. As mentionned in the theoretical analysis, for all of these communications I use *broadcasting*.

For sharing ants between nodes, we can note that each node computes the distribution. This is needed because each node has to know how much ants have the others to take the right random numbers from the file (to have a comparable result at the end of the algorithm with other implementations) :

```
int antsPerNode = totalNAnts / psize;
int restAnts = totalNAnts - antsPerNode * psize;

for (i = 0; i < psize; i++) {
  nAntsPerNode[i] = antsPerNode;
  if (restAnts > i) {
    nAntsPerNode[i]++;
  }
}

nAnts = nAntsPerNode[prank];

// Compute number of ants in nodes before me (prank smaller than mine)
int nAntsBeforeMe = 0;
for (i = 0; i < psize; i++) {
  if (i < prank) {
    nAntsBeforeMe += nAntsPerNode[i];
  } else {
    i = psize;
  }
}
```

About the algorithm itself, the implementation is strictly the same except that I can use internal and external loops to reduce communications (it is presented in the theoretical analysis). The algorithm looks like this one :

```
// Set the random counter to have right random values in each nodes
random_counter = (random_counter + (onNodeIteration * nAntsBeforeMe * nCities)) %
  nRandomNumbers;

while (external_loop_counter < externalIterations &&
  terminationCondition < (long) ceilf(externalIterations * onNodeIteration *
    terminationConditionPercentage)) {

  loop_counter = 0;
  while (loop_counter < onNodeIteration) {
    // Serial algorithm
    loop_counter++;
  }

  // Merging part
```

```
  external_loop_counter++;

  // Set the counter correctly for next random numbers on current node
  random_counter = (random_counter + (onNodeIteration * (totalNAnts - nAnts) *
    nCities)) % nRandomNumbers;
}
```

As not all improvements are good in term of result quality, I will present the three different ways I chose the do for merging informations between nodes. Thus, how communications are made will be presented in subsection 3.2.

Nevertheless, we can note here that, for sharing informations, each node needs to send to each other its best path of the iteration, but also its bestCost, its pheromons values on its best path (each node has its own local history) and the termination condition. Then, receiving nodes have to compare best path with their own and, if the received cost is better, the node updates its own best path, its best cost and its termination condition. The merge of pheromons depends on implementation (see subsection 3.2). We still can note that for merging pheromons, we always add the received pheromons value with our own and the we compute the average for each edge.

It is also important to know that the `terminationCondition` is commented in the provided code to be able to run the algorithm without it.

3.2 Optimality of results

To test the optimality of results, I implemented 3 different parallel algorithm and I tested them with and without the termination condition (stop after having the same result a certain number of iterations).

3.2.1 First parallel implementation

For the first implementation, I chose to share between nodes all best paths found on each node after a certain number of local iterations. To do that, I simply need to merge pheromons values received by all other nodes :

```
for (i = 0; i < psize; i++) {

  // Communications

  // If I am not node i, I will check if values from node i are better than mine
  if (prank != i) {
    if (otherBestCost < tempBestCost) {
      tempTerminationCondition = otherTerminationCondition;
      tempBestCost = otherBestCost;
      copyVectorInt(otherBestPath, tempBestPath,  nCities);
    } else if (otherBestCost == tempBestCost) {
      // If the best cost is the same as mine,
      // I simply update the termination condition counter
      tempTerminationCondition += otherTerminationCondition;
    }

    // Update pheromons received from other node
```

```
    for (j = 0; j < nCities - 1; j++) {
      pheromonsUpdate[getMatrixIndex(otherBestPath[j],otherBestPath[j+1],nCities)] +=
        1.0;
      pheromonsUpdate[getMatrixIndex(otherBestPath[j+1],otherBestPath[j],nCities)] +=
        1.0;
      pheromons[getMatrixIndex(otherBestPath[j],otherBestPath[j+1],nCities)] +=
        otherPheromonsPath[j];
      pheromons[getMatrixIndex(otherBestPath[j+1],otherBestPath[j],nCities)] +=
        otherPheromonsPath[j];
    }
    pheromonsUpdate[getMatrixIndex(otherBestPath[nCities-1],otherBestPath[0],nCities)]
      += 1.0;
    pheromonsUpdate[getMatrixIndex(otherBestPath[0],otherBestPath[nCities-1],nCities)]
      += 1.0;
    pheromons[getMatrixIndex(otherBestPath[nCities-1],otherBestPath[0],nCities)]
      += otherPheromonsPath[nCities - 1];
    pheromons[getMatrixIndex(otherBestPath[0],otherBestPath[nCities-1],nCities)]
      += otherPheromonsPath[nCities - 1];
  }
}
```

To test this code, I tried with one local iteration and with hundred local iterations and also with and without the termination condition (stop after we consider the result stable enough). In all cases, I did 5 executions with 5 different sets of random numbers. The figures show the costs for the serial implementation the parallel one with 1, 2, 4, 8 and 16 nodes. Other results can be found in the *result* folder.

All executions were made with 500 cities, 30'000 iterations, α and β equal to 1, 0.9 for evaporation coefficient and 32 ants[18].

One local iteration without termination condition (figures 17 and 18)

Best Cost	Serial	1	2	4	8	16
rand1	154496	154496	148475	142521	159332	143109
rand2	143087	143087	139158	106823	110784	127501
rand3	166698	166698	137681	134531	138969	137076
rand4	191951	191951	144172	115838	150429	147284
rand5	197345	197345	122878	152242	145672	145474

Figure 17: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	0	0	1	0
Parallel beats Serial	0	5	5	4	5

Figure 18: *Summary of measurements from previous figure (17)*

Hundred local iterations without termination condition (figures 19 and 20)

[18] In the *result* folder, you can find executions with less iterations or less ants. Indeed, I observed that if the number of iterations is not big enough compared to the problem size, the serial implementation has not enough time to converge compared to the parallel implementation using many nodes.

Best Cost	Serial	1	2	4	8	16
rand1	177592	177592	187736	247970	257124	244105
rand2	162085	162085	183043	233283	214107	278220
rand3	186004	186004	184833	195020	226533	231918
rand4	185873	185873	199568	176070	233620	241308
rand5	177384	177384	180036	203247	260322	282898

Figure 19: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	4	4	5	5
Parallel beats Serial	0	1	1	0	0

Figure 20: *Summary of measurements from previous figure (19)*

One local iteration with termination condition (figures 21 and 22)

I also tested to use another termination condition that is to have the same best cost after a certain number of iterations. For the results presented here (and below), the termination condition was of 70% of the total number of loops. We can observe that with this termination condition. the parallel algorithm converges really quickly, but the result is clearly bad and worst than the one from the serial implementation. Thus, I gave up this improvement.

Best Cost	Serial	1	2	4	8	16
rand1	178031	178031	1743417	1743417	1743417	1743417
rand2	180789	180789	1775744	1775744	1775744	1775744
rand3	152812	152812	1728234	1728234	1728234	1728234
rand4	188825	188825	1730664	1730664	1730664	1730664
rand5	161276	161276	1748895	1748895	1748895	1748895

Figure 21: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	5	5	5	5
Parallel beats Serial	0	0	0	0	0

Figure 22: *Summary of measurements from previous figure (21)*

Hundred local iterations with termination condition (figures 23 and 24)

Best Cost	Serial	1	2	4	8	16
rand1	181171	181171	1649680	1634016	1728163	1721699
rand2	179519	179519	1623298	1635800	1728501	1748994
rand3	190283	190283	1632833	1548522	1687993	1753994
rand4	146264	146264	1688079	1707766	1647953	1695028
rand5	192344	192344	1695841	1635615	1590794	1716914

Figure 23: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	5	5	5	5
Parallel beats Serial	0	0	0	0	0

Figure 24: *Summary of measurements from previous figure (23)*

3.2.2 Second parallel implementation

For the second implementation, I tried to share only the best path between all nodes at each external iteration. To do that, each node has still to share its best values but it keeps only the best received one (or its own if it has the best result). In term of code, the only impact is that the update of pheromons from others is done after the `for` loop and not in the loop (loop presented in subsection 3.2.1).

One local iteration without termination condition (figures 25 and 26)

Best Cost	Serial	1	2	4	8	16
rand1	154496	154496	154576	125320	137840	145875
rand2	143087	143087	134842	165693	140243	126733
rand3	166698	166698	149265	149338	123710	166576
rand4	191951	191951	155158	140697	164721	161000
rand5	197345	197345	145260	173973	112334	147570

Figure 25: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	1	1	0	0
Parallel beats Serial	0	4	4	5	5

Figure 26: *Summary of measurements from previous figure (25)*

Hundred local iterations without termination condition (figures 27 and 28)

Best Cost	Serial	1	2	4	8	16
rand1	177592	177592	187736	247970	257124	244105
rand2	162085	162085	183043	233283	214107	278220
rand3	186004	186004	184833	195020	226533	231918
rand4	185873	185873	199568	176070	233620	241308
rand5	177384	177384	180036	203247	260322	282898

Figure 27: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	4	4	5	5
Parallel beats Serial	0	1	1	0	0

Figure 28: *Summary of measurements from previous figure (27)*

One local iteration with termination condition (figures 29 and 30)

Best Cost	Serial	1	2	4	8	16
rand1	178031	178031	1743417	1743417	1743417	1743417
rand2	180789	180789	1775744	1775744	1775744	1775744
rand3	152812	152812	1728234	1728234	1728234	1728234
rand4	188825	188825	1730664	1730664	1730664	1730664
rand5	161276	161276	1748895	1748895	1748895	1748895

Figure 29: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	5	5	5	5
Parallel beats Serial	0	0	0	0	0

Figure 30: *Summary of measurements from previous figure (29)*

Hundred local iterations with termination condition (figures 31 and 32)

Best Cost	Serial	1	2	4	8	16
rand1	181171	181171	1649680	1634016	1728163	1721699
rand2	179519	179519	1623298	1635800	1728501	1748994
rand3	190283	190283	1632833	1548522	1687993	1753994
rand4	146264	146264	1688079	1707766	1647953	1695028
rand5	192344	192344	1695841	1635615	1590794	1716914

Figure 31: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	5	5	5	5
Parallel beats Serial	0	0	0	0	0

Figure 32: *Summary of measurements from previous figure (31)*

3.2.3 Third parallel implementation

For the third implementation, I chose to share all the pheromons matrix instead of only the values from the best path. The code becomes simple, because you just have to send the matrix in one message :

```
if (MPI_Bcast(&otherPheromons[0], nCities * nCities, MPI_DOUBLE, i, MPI_COMM_WORLD)
  != MPI_SUCCESS) {

  printf("Node %d : Error in Broadcast of otherPheromons", prank);
  MPI_Finalize();
  return -1;
}
```

Of course, there will be a lot of communications for nothing (the majority of the matrix will have low values close to 0).

One local iteration without termination condition (figures 33 and 34)

Best Cost	Serial	1	2	4	8	16
rand1	154496	154496	176922	147007	164259	159278
rand2	143087	143087	140698	157945	150729	160835
rand3	166698	166698	149420	186936	167434	162441
rand4	191951	191951	177700	156597	142280	160030
rand5	197345	197345	199938	139334	166256	162379

Figure 33: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	2	2	3	2
Parallel beats Serial	0	3	3	2	3

Figure 34: *Summary of measurements from previous figure (33)*

Hundred local iterations without termination condition (figures 35 and 36)

Best Cost	Serial	1	2	4	8	16
rand1	177592	177592	187736	247970	257124	244105
rand2	162085	162085	183043	233283	214107	278220
rand3	186004	186004	184833	195020	226533	231918
rand4	185873	185873	199568	176070	233620	241308
rand5	177384	177384	180036	203247	260322	282898

Figure 35: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	4	4	5	5
Parallel beats Serial	0	1	1	0	0

Figure 36: *Summary of measurements from previous figure (35)*

One local iteration with termination condition (figures 37 and 38)

Best Cost	Serial	1	2	4	8	16
rand1	178031	178031	1743417	1743417	1743417	1743417
rand2	180789	180789	1775744	1775744	1775744	1775744
rand3	152812	152812	1728234	1728234	1728234	1728234
rand4	188825	188825	1730664	1730664	1730664	1730664
rand5	161276	161276	1748895	1748895	1748895	1748895

Figure 37: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	5	5	5	5
Parallel beats Serial	0	0	0	0	0

Figure 38: *Summary of measurements from previous figure (37)*

Hundred local iterations with termination condition (figures 39 and 40)

Best Cost	Serial	1	2	4	8	16
rand1	181171	181171	1649680	1634016	1728163	1721699
rand2	179519	179519	1623298	1635800	1728501	1748994
rand3	190283	190283	1632833	1548522	1687993	1753994
rand4	146264	146264	1688079	1707766	1647953	1695028
rand5	192344	192344	1695841	1635615	1590794	1716914

Figure 39: *Measurements for 5 executions with one local iteration, 30'000 external iterations, 500 cities, α and β equal to 1, evaporation coefficient to 0.9 and 32 ants.*

Summary	1	2	4	8	16
Serial beats Parallel	0	5	5	5	5
Parallel beats Serial	0	0	0	0	0

Figure 40: *Summary of measurements from previous figure (39)*

3.2.4 Comments

Watching these results, we first observe that the additional termination condition is clearly not a good idea. Indeed, in all executions with this condition, the parallel implementation converges too quickly and stays stuck in a bad solution.

The other improvement (local iterations) is also not a good idea. Indeed, we can observe that the results are in general not so far from the serial execution but they are generally beaten by it.

Finally, the execution with one local iteration seems to be the good solution to solve this problem. Indeed, we can conclude that sharing informations after each iterations makes the parallel algorithm good in term of optimality, but of course, it increases communications and the parallelization is not as good as expected initially.

About the different parallel implementations, the only interesting thing is that the third implementation (sharing whole pheromon matrix) is not as good as first and second implementation. Indeed, for the first and second implementations, the solution is often better than the serial algorithm while the third implementation (when everything is shared) is as good as the serial implementation. Thus, I am sure that third algorithm is a good one. For the first and second, when you observe the costs for the best paths, they are not so much better than the serial one. I conclude that having less informations (nodes do not share all the pheromon's matrix) makes them giving less importance to already not important paths.

Thus, these two parallel implementations are giving good results. I guess that they do not stay stucked in a local minimum because of the maximum values of pheromons paths. Indeed, when you observe results with the termination condition, parallel algorithms are always worst because they are quickly stucked in a local minimum. Limiting the maximum value of pheromons will leave time to better paths to have higher pheromon's values on them and therefore to go out of the suboptimal solution.

3.3 Speedups

For the different speedups, we can note that for all tests I have made, the speedup always looks like the same. Thus, I will only present one relative and one absolute speedup for one execution of my program, knowing that the others are similar.

We can note that I will not show here the speedups for execution with the additional termination condition because the results were clearly not optimal and also not the speedups for hundred iterations, for the same reason. Nevertheless, they can be found directly in the `results` directory.

The following speedups are the ones from file *rand1* from figure 17.

3.3.1 Relative Speedup

First parallel implementation (figure 41)

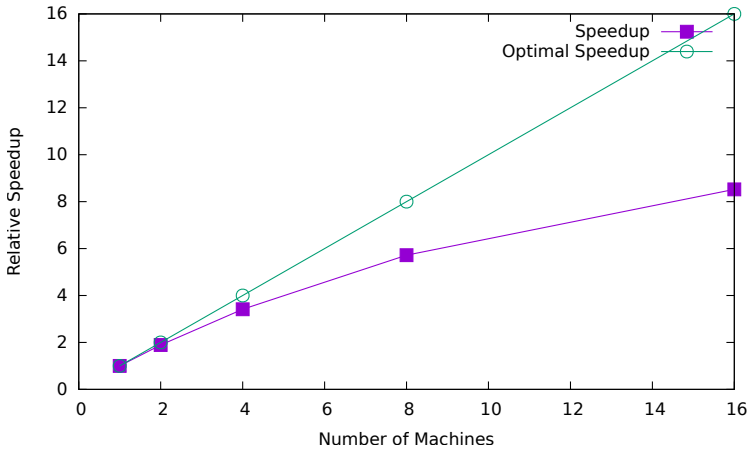


Figure 41: *Parallel algorithm 1 - Relative speedup for random file 1 from results in figure 17*

	Serial Time	1	2	4	8	16
Execution Time	11020.097182	11498.497093	6082.637220	3370.134546	2011.000059	1348.943900
Relative Speedup		1.000000	1.890380	3.411880	5.717800	8.524073

Figure 42: *Parallel Algorithm 2 - Times for each execution in ms and speedups.*

Second parallel implementation (figure 43)

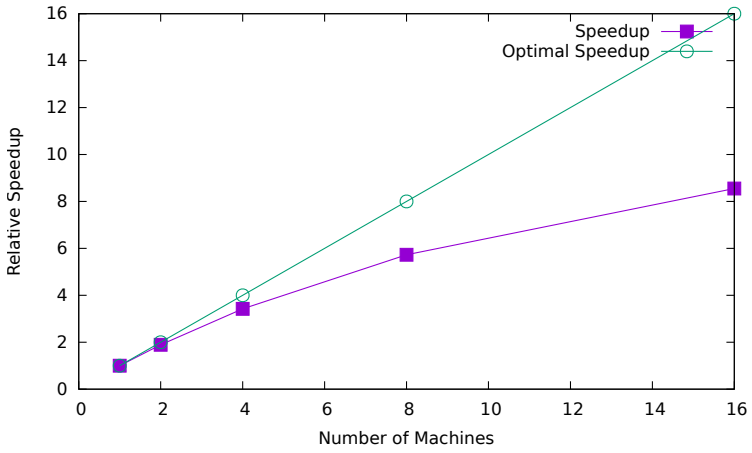


Figure 43: *Parallel algorithm 2 - Relative speedup for random file 1 from results in figure 17*

	Serial Time	1	2	4	8	16
Execution Time	11020.097182	11525.187912	6087.934365	3369.249516	2013.014723	1347.747293
Relative Speedup		1.000000	1.893119	3.420698	5.725337	8.551445

Figure 44: *Parallel Algorithm 2 - Times for each execution in ms and speedups.*

Third parallel implementation (figure 45)

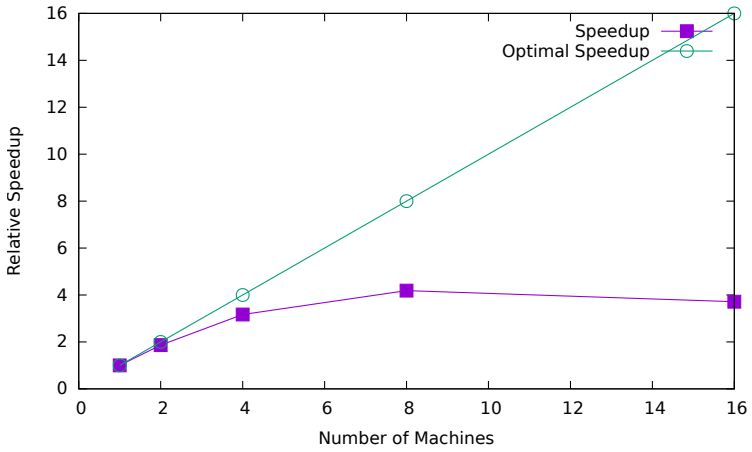


Figure 45: *Parallel algorithm 3 - Relative speedup for random file 1 from results in figure 17*

	Serial Time	1	2	4	8	16
Execution Time	11020.097182	11536.970434	6187.463539	3640.974301	2756.442646	3106.276513
Relative Speedup		1.000000	1.864571	3.168649	4.185456	3.714083

Figure 46: *Parallel Algorithm 3 - Times for each execution in ms and speedups.*

3.3.2 Absolute Speedup

First parallel implementation (figure 47)

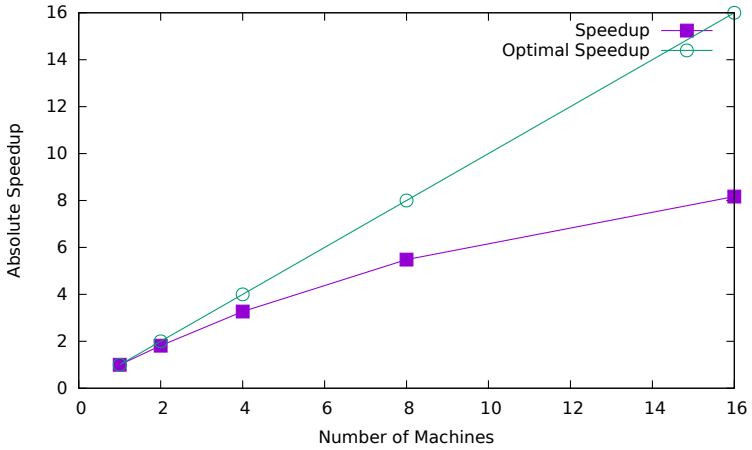


Figure 47: *Parallel algorithm 1 - Absolute speedup for random file 1 from results in figure 17*

	Serial Time	1	2	4	8	16
Execution Time	11020.097182	11498.497093	6082.637220	3370.134546	2011.000059	1348.943900
Absolute Speedup		0.958394	1.811730	3.269927	5.479908	8.169425

Figure 48: *Parallel Algorithm 1 - Times for each execution in ms and speedups.*

Second parallel implementation (figure 49)

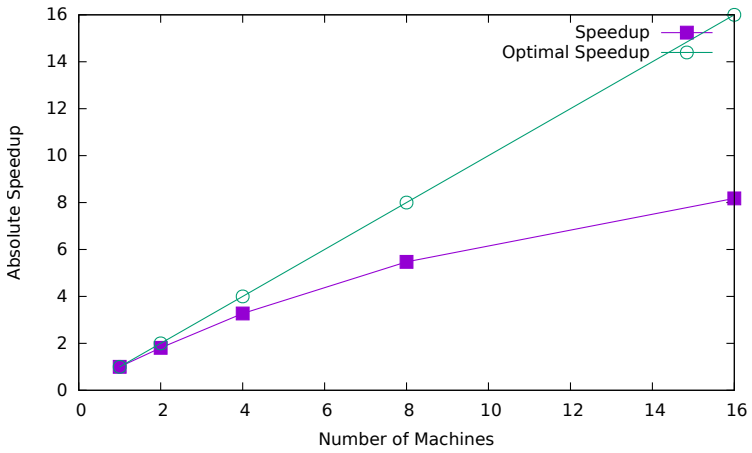


Figure 49: *Parallel algorithm 2 - Absolute speedup for random file 1 from results in figure 17*

	Serial Time	1	2	4	8	16
Execution Time	11020.097182	11525.187912	6087.934365	3369.249516	2013.014723	1347.747293
Absolute Speedup		0.956175	1.810153	3.270786	5.474424	8.176679

Figure 50: *Parallel Algorithm 2 - Times for each execution in ms and speedups.*

Third parallel implementation (figure 51)

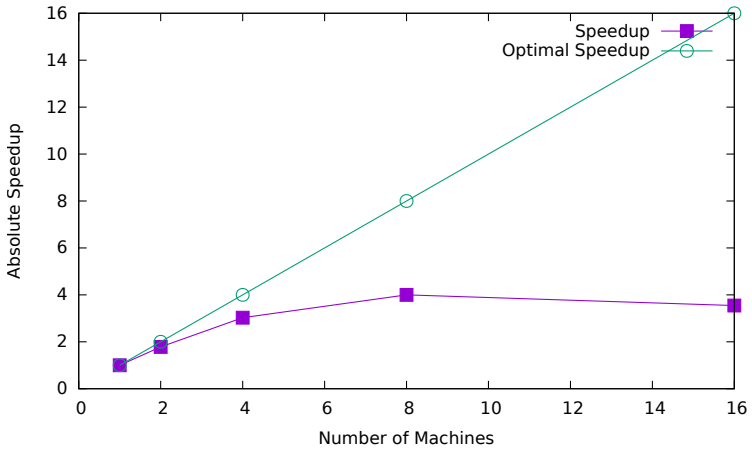


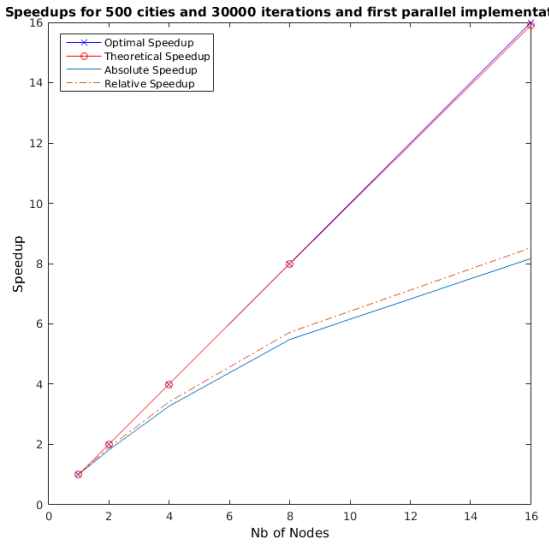
Figure 51: *Parallel algorithm 3 - Absolute speedup for random file 1 from results in figure 17*

	Serial Time	1	2	4	8	16
Execution Time	11020.097182	11536.970434	6187.463539	3640.974301	2756.442646	3106.276513
Absolute Speedup		0.955198	1.781036	3.026689	3.997941	3.547687

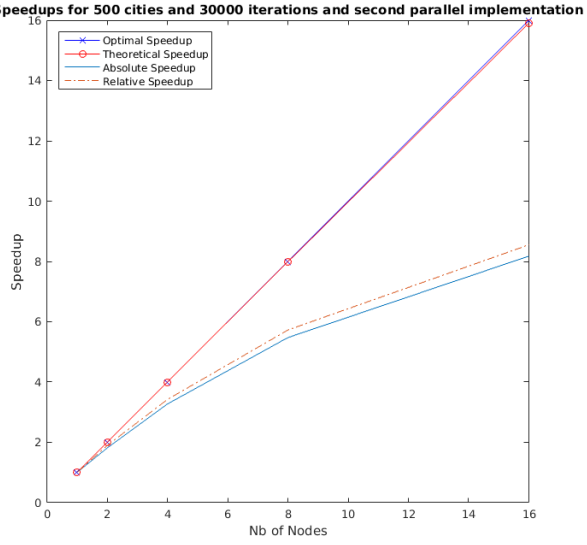
Figure 52: *Parallel Algorithm 3 - Times for each execution in ms and speedups.*

3.3.3 Comparison with Theoretical Speedup

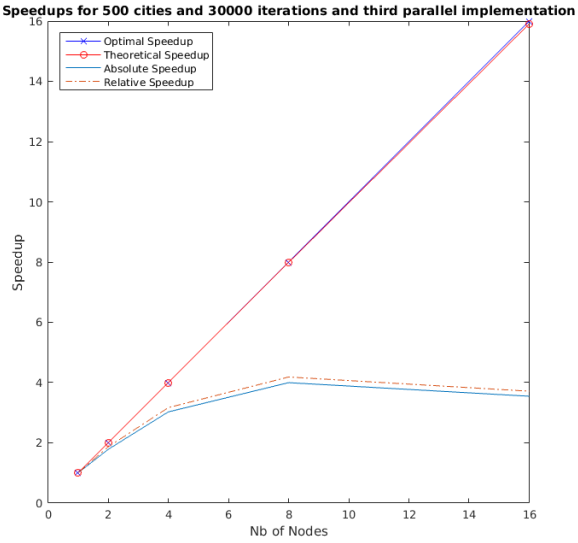
Figure 53 shows for each parallel implementation the absolute, the relative and the theoretical speedup graph.



(a) *Speedups for first parallel implementation.*



(b) *Speedups for second parallel implementation.*



(c) *Speedups for third parallel implementation.*

Figure 53: *All speedups in one graph for each parallel implementation.*

3.3.4 Comments

We can observe in these graphs that, as expected, the theoretical speedup is clearly too much optimistic. Indeed, the main reasons why the speedups are not so good is that communications take more time than expected and we must send more than only the best path to be able to process the algorithm correctly.

Between the parallel algorithms themselves, we observe that the third implementation as clearly the worst speedup of all implementations. This is totally normal as we send all the pheromon's matrix in this implementation. Therefore, there are more communications to do and thus they take more time when there are many nodes.

We also can observe that, with 16 nodes, the third implementation has a worst speedup than with only

8 nodes. It indicates us this algorithm does not scale well.

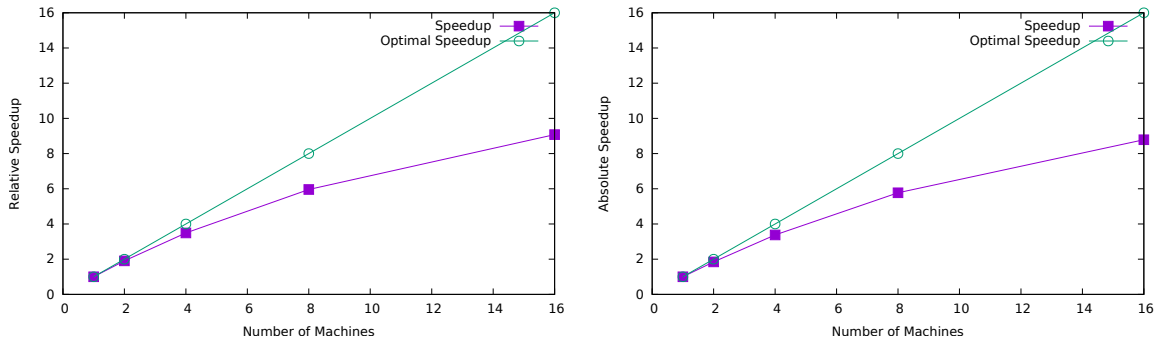
About solutions that are not optimal[19], we can observe that speedups are really good thanks to improvements (local iterations, termination condition). Indeed, I tried to reduce communications and it works. Speedups are really good (even better than good with the additional termination condition), but as the optimality of the result is not guaranteed, these solutions are not interesting.

3.3.5 Other measurements

Below, you can find different experiments with different problem sizes. Values used to create these graphs can be found on the DVD attached to this report.

All these experiments were made with only one iteration locally and without the termination condition. The changed arguments are the size of the city, the number of ants and the maximum number of iterations.

250 cities with 32 ants and 15'000 iterations (first algorithm)



(a) *Relative Speedup* (b) *Absolute Speedup*

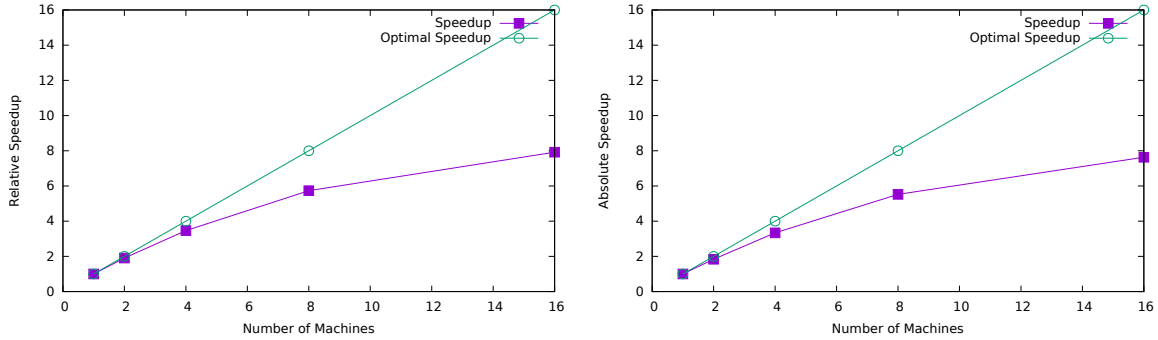
Figure 54: *First parallel implementation - Speedups for 250 cities, 32 ants and 15'000 iterations.*

Serial beats Parallel	0	0	0	1	0
Parallel beats Serial	0	5	5	4	5

Figure 55: *Summary of measurements comparison between serial and parallel implementation for 250 cities, 32 ants and 15'000 iterations.*

100 cities with 32 ants and 15'000 iterations (first algorithm)

[19] You can find them attached to this report.



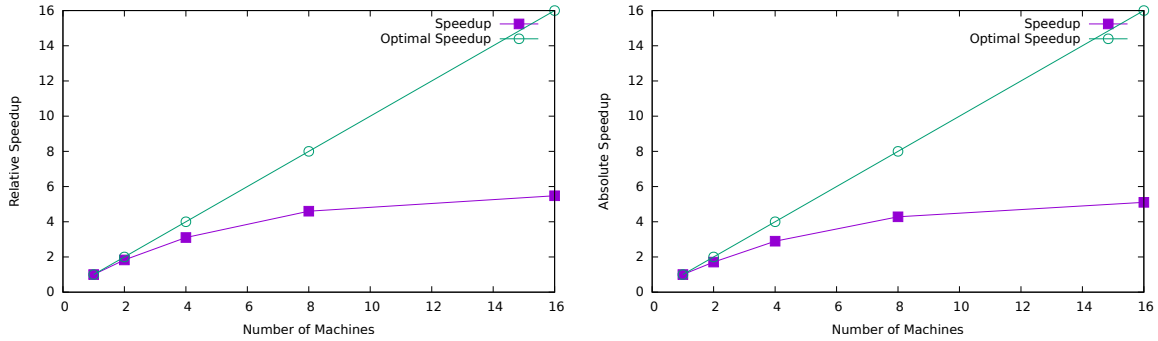
(a) *Relative Speedup* (b) *Absolute Speedup*

Figure 56: *First parallel implementation - Speedups for 250 cities, 32 ants and 15'000 iterations.*

Serial beats Parallel	0	1	1	2	2
Parallel beats Serial	0	4	4	3	3

Figure 57: *Summary of measurements comparison between serial and parallel implementation for 100 cities, 32 ants and 15'000 iterations.*

100 cities with 16 ants and 15'000 iterations (first algorithm)



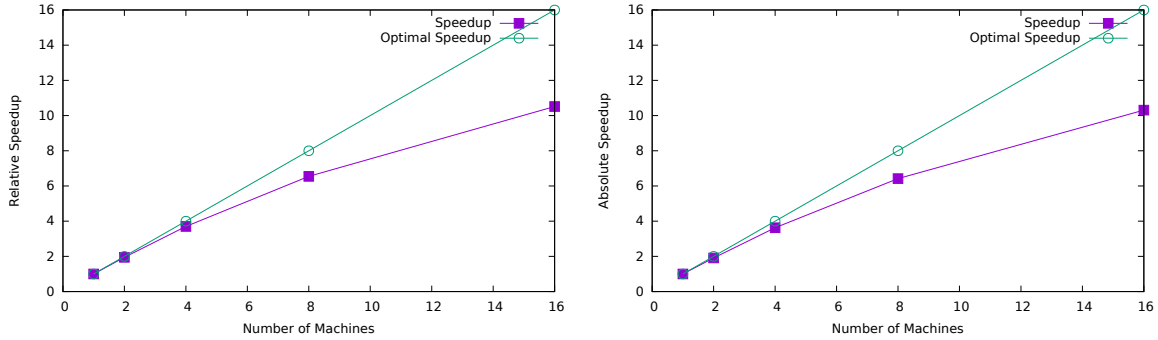
(a) *Relative Speedup* (b) *Absolute Speedup*

Figure 58: *First parallel implementation - Speedups for 100 cities, 16 ants and 15'000 iterations.*

Serial beats Parallel	0	3	2	3	2
Parallel beats Serial	0	2	3	2	3

Figure 59: *Summary of measurements comparison between serial and parallel implementation for 100 cities, 16 ants and 15'000 iterations.*

100 cities with 64 ants and 15'000 iterations (first algorithm)



(a) *Relative Speedup* (b) *Absolute Speedup*

Figure 60: *First parallel implementation - Speedups for 100 cities, 64 ants and 15'000 iterations.*

Serial beats Parallel	0	2	1	2	2
Parallel beats Serial	0	3	4	3	3

Figure 61: *Summary of measurements comparison between serial and parallel implementation for 100 cities, 64 ants and 15'000 iterations.*

3.3.5.1 Comments With these results, we can observe several things. Indeed, in terms of optimality, the experiments made in section 3.2 have shown that first and second implementation of the parallel algorithm have always beaten serial implementation. Here, with smaller problems, we observe that the first algorithm is as good as the serial one (figure 59 for example). Thus, in section 3.2, the serial implementation did not have the time to obtain a really good result.

To avoid this problem, we simply need to do more iterations, but as we observed in this report, the parallel implementation converges faster to an optimum. The termination condition could be here something good to really improve the performances of the parallel algorithm. Nevertheless, this termination condition should be carefully chosen because we observed that there is a huge risk that the parallel implementation stays stucked in a suboptimal solution (as observed in this report).

We also see that the number of ants is important to have a good speedup, as claimed in section 2.2. Indeed, with 16 ants (figure 58), the speedup is worst than with 64 (figure 60).

Thus, we can conclude that, to have a good speedup, the problem has to be huge and we need to have a lot of ants compared to the number of nodes.

4 Conclusion

In conclusion, we can say that this problem scales relatively well even if we must keep a lot of communications between nodes. Indeed, if we select carefully the number of nodes and the number of ants, we can have a good speedup.

We also saw that to have an optimal result, we need to be careful with arguments given to the program. The choice of the map size and the number of ants strongly impacts the number of iterations to do to be sure to have an optimal results. If the number of iterations is too low, the program will not have the time to find the optimal solution. If we give too much iterations, we will kill the speedup and loose all the gain of parallelizing this problem. Thus, the choice of the number of iterations is delicate.

We also can notice that other arguments (α and β) are also important for this algorithm. Indeed, to improve its efficiency, we can implement an algorithm where these values evolves during the execution to give more importance to distance at the start of the algorithm and to privilege pheromons at the end. This improvement can also helps to obtain a good result easily and thus to improve the parallel implementation.