A Parallel Implementation of an Asynchronous Team to the Point-to-Point Connection Problem

Ricardo C. Corrêa [*,1] Fernando C. Gomes [*,1]
Carlos A.S. Oliveira [**] Panos M. Pardalos [**]

Abstract

We propose a parallel and asynchronous approach to give near-optimal solutions to the non-fixed point-to-point connection problem. This problem is NP-hard and has practical applications in multicast routing. The technique adopted to solve the problem is an organization of heuristics that communicate with each other by means of a virtually shared memory. This technique is called A-Teams (for Asynchronous Teams). The virtual shared memory is implemented in a physically distributed memory system. Computational results comparing our approach with a branch-and-cut algorithm are presented.

Key words: Point-to-point connection problem, multicast routing, parallel algorithms, asynchronous models, heuristics

1 Introduction

Multicast routing is the sending of information from a number of sources for multiple receivers in a network, with a simple "transmit" operation [1]. Nowadays, multicast routing is required in many situations, including video-on-demand, video-conference, radio stations on Internet, and delivery of soft-

[*] Departamento de Ciência da Computação, Universidade Federal do Ceará (UFC), Fortaleza, CE, Brazil.
[**]Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL, USA.
Email addresses: `correa@lia.ufc.br` (Ricardo C. Corrêa), `carvalho@lia.ufc.br` (Fernando C. Gomes), `oliveira@ufl.edu` (Carlos A.S. Oliveira), `pardalos@ufl.edu` (Panos M. Pardalos).
[1] Supported in part by the CNPQ (Brazilian Research Council).

ware. In such situations, three types of nodes can be distinguished in a network, namely *source nodes*, from which originates the data; *destination nodes*, which receive the data; and *link nodes*. All source, destination and link nodes can be used to route information from sources to destinations. Thus, depending on the structure of the network, there is a large number of possibilities to route information from source to destination nodes. Consequently, optimization problems may be formulated in relation to these many possible routings.

In this paper, we concentrate our attention in one of such problems, the point-to-point connection (PPC) problem. We present a parallel asynchronous methodology to solve it, called Asynchronous Teams (A-Teams). We also demonstrate an implementation of this methodology and give computational results of our algorithm. This paper is organized as follows. In Section 2 we give a description of the point-to-point connection problem. We also discuss some of the complexity results for the PPC problem and its variants. In Section 3 the Asynchronous Teams method is discussed in detail. We present our polynomial complexity algorithm to the PPC problem in Section 4. In Section 5 a parallel implementation of the Asynchronous Teams is discussed. In Section 6, computational results found using the algorithm are presented and analyzed. Finally, in Section 7 we present conclusions and remarks on future research.

2 The Point-to-Point Connection Problem

In this section we define the Point-to-point Connection (PPC) problem. This problem occurs as a kind of multicast routing problem, as well as in other areas, like VLSI and vehicle routing [2]. From the optimization point of view, we need to characterize the solutions and an objective function. Let $G(V, E)$ be a graph where V represents the set of devices connected in a network and E represents the physical links (cables, satellite connections) between these devices. We assign a cost $w(i, j)$ to each link (i, j) in the network. This quantity can represent real costs of using the link. It can also model the time required for delivering a message through the link. We assume that costs of network links are fixed (a situation corresponding to static routing). In addition, we assume that every link has the necessary throughput to convey the desired data.

Moreover, we assume that the number of source and destination nodes are identical. Situations in which the number of sources is less than that of receivers can be easily handled if we include dummy source nodes into the network.

A solution is characterized by a forest $E' \subset E$, composed of a set of paths

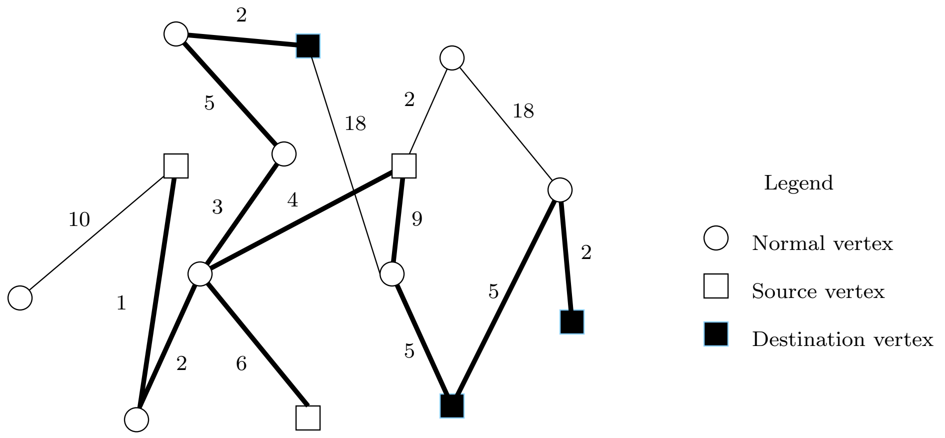

Fig. 1. Instance of the PPC. The thick lines represent the best solution, with value 44.

joining the source to the destination nodes (see Figure 1). The cost of a solution is the sum of the costs of all links involved in this solution. Then, we want to find a way to link the source and destination nodes with minimum total cost.

In mathematical terms, let $|V| = n$ and $|E| = m$. For a given positive integer p, define $S \subset V$ and $D \subset V$ to be the set of source and destination nodes, respectively, where $|S| = |D| = p$ and $S \cap D = \emptyset$. A *source-destination coupling* (or *coupling*, for short) is a one-to-one mapping from S to D. Then a *point-to-point connection*, referred to simply as *connection*, is a subset E' of E such that there is a coupling in which each source-destination pair is connected by a path containing only edges in E'. Notice that these paths may intersect. Denote the cost of E' by $f(E') = \sum_{(u,v)\in E'} w(u,v)$. The objective in the undirected and non-fixed *point-to-point connection problem (PPCP)* is to find a minimum cost point-to-point connection.

We can devise four different variants of the PPC problem. Besides the undirected, non-fixed version defined above, we can make G directed or fix destinations. In the latter case, a fixed coupling is given as input. Li *et al.* [2] have proved that all four versions of the problem are NP-hard. In the same work, the authors gave a dynamic programming algorithm with time complexity $O(n^5)$ for the fixed destinations case of the problem on directed graphs when $p = 2$. More recently, Natu and Fang [3] proposed another dynamic programming algorithm, still for $p = 2$, with overall time complexity $O(mn + n^2 \log n)$. They also presented an algorithm for the fixed destinations case when the graph is directed and $p = 3$. This algorithm has time complexity $O(n^{11})$. Goemans and Williamson [4] present an approximation algorithm for the PPC problem, that runs in $O(n^2 \log n)$ and gives its results within a factor of $2 - 1/p$ of an optimal solution. In the case in which $p = 1$, it is easy to see that the problem is equivalent to the shortest path problem.

2.1 Mathematical programming formulation

To derive a mathematical programming model of the PPCP, let us define the following variables:

$$x_{i,j} = \begin{cases} 1 \text{ if } (i,j) \in E' \\ 0 \text{ otherwise} \end{cases}$$

The PPCP can be stated as:

$$\textbf{min} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j} w_{i,j}$$

such that

$$\sum_{i \in A} \sum_{j \notin A} x_{i,j} \geq 1 \quad \forall A \subset V(G), \text{ whenever } |A \cap S| \neq |A \cap D|$$

$$x_{i,j} \in \{0,1\} \text{ for all } i,j = 1,\ldots,n$$

where S, D are respectively the source and destination sets discussed above. Although there exist polynomial time dynamic programming algorithms for solving point-to-point connection instances, they only apply when the number of source-destination pairs is no greater than 3. In real-world applications we need to solve far larger problem instances, with many source-destination pairs. In this paper, we are concerned with providing a near-optimal solution to instances of the undirected, non-fixed PPC problem in an acceptable amount of computation time. We design some simple heuristics that cooperate to produce good solutions. These heuristics are organized as an Asynchronous Team (A-Team), as show in the next section.

3 A-Team algorithms

An Asynchronous Team (A-Team) is an organization of heuristics that communicate with each other exchanging solutions through a shared memory. Each heuristic can make its own choices about inputs, scheduling and resource allocation. This method was proposed by Souza [5] and has been applied successfully to the Traveling Salesman Problem [5], Set Covering Problem [6], Job Shop Scheduling Problem [7,8] and Constraint Satisfaction Problem [9].

The simple ideas around the A-Teams method make it suitable to be used to solve combinatorial optimization problems. This characteristic becomes more interesting when there are only few heuristics for a problem. In this case, heuristics can be easily combined in an A-Team to give improved results. Even for problems for which there are good known heuristics, this method can be used to combine the strengths of different strategies. This section is devoted to the definition of A-Team algorithms. The next section contains details about its implementation for solving the PPC problem.

An A-Team can be defined as a set $M = \{M_1, M_2, \dots, M_l\}$ of shared memories and a set $H = \{H_1, H_2, \dots, H_k\}$ of heuristics. Each memory M_i holds a set of potential solutions $S_i = \{s_1, s_2, \dots, s_{k_i}\}$. Each heuristic H_j has access (reading, writing, or read-writing) to a subset $M_l, l \in R(j)$ of the memories, where $R(j) \subset \{1, \dots, k\}$ is the *reach* of heuristic H_j. The heuristics H work as asynchronous strategies, capable of reading and writing new solutions to and from its associated memories. The general idea is that the memory holds a population of solutions, which is modified by heuristics. Each heuristic can be very simple, but must contribute different features to the generated solutions. The content of memories can therefore evolve in the sense that the objective function of the best solution is improving.

Among the set H of heuristics we can identify some with specific attributes. For example, *construction heuristics* work (alone or together with other heuristics) to create a new solution. They are used mainly to fill the memories at the beginning of processing. *Improvement heuristics* are used to modify existing solutions. An improvement heuristic reads a solution from any of the memories to which it has access, executes an improvement procedure and then writes the resulting solution again to one of the accessible memories. *Destruction heuristics* act every time that it is necessary to discard a solution from memory. This happens whenever a new solution must be written in an already full memory. The flow of execution of an A-Team is therefore inherently parallel. A generically equivalent sequential version is presented in Algorithm 3.

In Figure 2 a simple A-Team is presented, where arrows represent heuristics and the rectangle represents the only shared memory $M = M_1$. One of the heuristics is marked as a constructor, which is represented differently from other heuristics. The destructor also has its own representation. In this case, the memory M_1 is shared by all heuristics in H. In any A-Team, the operations that the heuristics are allowed to perform on M are two, namely *reading* and *writing*. The reading operation does not take any input, and returns a solution stored in M, selected with some arbitrary criterion. The input of the writing operation is a solution, which is possibly written in M according also to an arbitrary criterion. If a solution is to be written, then another solution in M is selected to be deleted and replaced by the new solution.

```
for each of the memories $M_i \in M$ do
    for $j \leftarrow 1$ to $k_i$ do
        Call one of the constructors to create solution $s_j$
        write($s_j, M_i$)
    end for
end for
while $termination\_condition \neq$ True do
    $H_i \leftarrow$ random($H$)
    $l \leftarrow$ random($R(i)$)
    $s_j \leftarrow$ read_solution($M_l$)
    $s_k \leftarrow H_i(s_j)$
    {Call a destructor to remove a solution from $M_l$}
    destructor($M_l$)
    {Write the resulting solution}
    write($s_k, M_l$)
    {Update best solution}
    if $f(s_k) < f$(best-solution) then
        best-solution $\leftarrow s_k$
    end if
end while
```

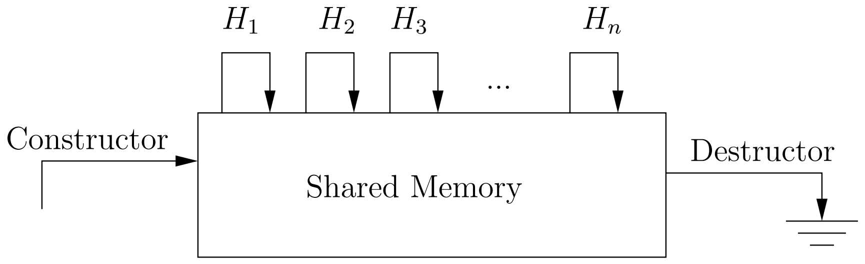

Fig. 2. Example of an A-Team. Arrows represent heuristics and the rectangle represents shared memory.

We characterize A-Team algorithms predominantly by the following three features. First, the heuristics are *autonomous*, which means that each of them is able to run independently from the others. Second, the shared memory runs *asynchronous* communications, which allows concurrent reading and writing operations from distinct heuristics without any synchronization among them. Finally, heuristics retrieve, modify, and store information continuously in the shared memory until a certain termination condition is achieved.

An execution of an A-Team can be described by a set of events. Each *event* is a 5-tuple composed by the following elements: the time at which the event occurs; the input data, if any; the state of the shared memory prior to the occurrence of the event; the state of the shared memory after the occurrence of the event; and the output data, if any. According to this formalism, there is a restricted number of types of events occurring during an A-Team execution:

- The initialization of global variables and the shared memory. This type of event occurs just once at the beginning, and takes no input. The state prior to the event is undefined, the state after the event contains the initial state of the global variables and the shared memory. No output is generated.
- The reading operation. Again, this type of event takes no input. The state does not change, and the output is a solution in M.
- The execution of a heuristic H_i. It takes one solution in $M_j, j \in R(i)$ as input and possibly changes the state of the global variables. The output of this type of event is also a solution.
- The writing operation. A solution is taken as input to be written in M. This may change the state of the solutions memory, but no output is generated.

Finally, the termination of the A-Team is decided by the algorithm `termination-condition`, as shown in Algorithm 3.

It is clear from the previous observations that there exist several distinct executions for a given A-Team. The solution obtained in each execution may differ. An execution is said to be *acceptable* if it leads to a near-optimal solution. Again, an acceptable execution of an A-Team is not unique. Thus, an acceptable execution is *efficient* if it is performed fast. Finally, an A-Team algorithm is considered *efficient* if it has efficient executions and these occur with high probability. In the next sections, we show in details a parallel A-Team algorithm for the PPC problem, and we check its efficiency experimentally.

4 Solving the PPC problem

In this section we describe the variables and heuristics of our A-Team for the PPC problem. The heuristics are divided in classes. We use *construction* heuristics (C*) that build a feasible solution from another feasible solution, *improvement* heuristics (I*) that try to improve a given feasible solution, and *consensus based* heuristics (CB*) that take sets of edges from two solutions of memory and construct a partial (unfeasible) solution based on them. Finally, in order to avoid memory overload, after each construction, improvement or consensus based heuristic run, a destruction heuristic (D) is used to remove a solution from M.

4.1 Global data structures

Some global variables are used for all heuristics. The matrix A stores the length of a shortest path between all pairs of nodes in G. The element $A_{u,v}$ indicates the length of a shortest path between u and v in G, for all $u, v \in V(G)$. A shortest path is stored in the matrix $\Pi = (\Pi_{u,v})$. The element $\Pi_{u,v}$ is the predecessor of v in some shortest path between u and v in G, for all $u, v \in V(G)$. The initial values of variables $A_{u,v}$ and $\Pi_{u,v}$ are calculated with the Floyd-Warshall's shortest paths algorithm in time $O(n^3)$ [10]. The variable *best_sol* stores the solution found so far with smallest cost. The objective function of the solution with greatest cost is stored in $worst\_sol$.

The shared memories M_i, $1 \le i \le m$ consist of finite linear lists of k_i solutions. The `read` operation does not take any input, and returns the solution stored in a position of M chosen at random. Initially, M is filled with connections obtained with the construction heuristic C1, described bellow. On the other hand, the input of the `write` operation is a solution s. The algorithm `write` initially tests if $f(s) < f(worst\_connection)$ and $s \neq s', \forall s' \in S$. If this is false, s is discarded and the writing operation is aborted. Otherwise, the "roulette wheel" method is used to randomly select a position p_i, $1 \le p_i \le k_i$, according to the value of the objective function for the solutions in M_i. Then, the solution s is assigned to $M_i[k]$.

4.2 Heuristic strategies

In what follows, we describe simple heuristics used to solve the PPC problem. These heuristics form the H set of agents operating on the shared memories. In the following description we denote by $u \rightsquigarrow v$ a path from vertex u to vertex v.

Construction Heuristic C1 This heuristic creates a completely random solution.

```
 1: $D \leftarrow$ set of destinations; $S \leftarrow$ set of sources
 2: $sol \leftarrow \emptyset$
 3: $i \leftarrow 1$
 4: while $D \neq \emptyset$ do
 5:     $d \leftarrow$ random destination in $D$
 6:     $D \leftarrow D \setminus d$
 7:     $path \leftarrow$ random path in $G$ from $S_i$ to $d$
 8:     $sol \leftarrow sol \cup path$
 9:     $i \leftarrow i + 1$
10: end while
```

```
11: return $sol$
```

Time complexity: $O(pm)$

Construction Heuristic C2 This heuristic takes a set E' of edges from the incomplete solution created by **CB1** or **CB2** (see bellow), and creates a complete solution as follows:

```
1: $sol \leftarrow \emptyset$
2: for each edge $(u, v) \in E'$ do
3:     Get a random source-destination pair $(a, b)$.
4:     $p_1 \leftarrow$ shortest-path$(a, u)$; $p_2 \leftarrow$ shortest-path$(v, b)$
5:     $p \leftarrow (p_1, p_2)$
6:     $sol \leftarrow sol \cup p$
7: end for
8: if $|E'| < p$ then
8:     connect remaining source-destination pairs using C1.
9: end if
```

Time complexity: $O(m)$

Construction Heuristic C3 This heuristic constructs a solution s in the same way as C1. The difference is that it guarantees that the final solution has no sources or destinations inside each path $(a, b) \in s$. This makes the paths in s completely disjoint. Time complexity for C3 is also $O(pm)$.

Consensus Based CB1 Takes the common edges of two solutions and inserts them in an auxiliary memory. This algorithm works together with heuristic **C2** to form a new feasible solution.

```
1: Select solutions $s_1$ and $s_2$ from memory $M$.
2: $E \leftarrow$ edges in $s_1 \cup s_2$
3: store $E$ in an auxiliary memory.
```

Time complexity: $O(m \log m)$

Consensus Based CB2 Compares two solutions s_1 and s_2 and stores the edges appearing only in the best of s_1, s_2.

```
1: Select solutions $s_1$ and $s_2$ from memory $M$.
2: if $f(s_1) \geq f(s_2)$ then
3:     $s'_1 \leftarrow s_1$; $s'_2 \leftarrow s_2$
4: else
5:     $s'_1 \leftarrow s_2$; $s'_2 \leftarrow s_1$
6: end if
7: Let $E \leftarrow$ edges of $s'_1 \setminus s'_2$
8: store $E$ in an auxiliary memory.
```

Time complexity: $O(m \log m)$

Improvement heuristic I1 This heuristic uses the edge with lowest weight to concatenate shortest paths in a solution.

```
1: Select a solution $s$ from memory $M$.
2: $p \leftarrow a \rightsquigarrow b$, for $(a, b) \in s$.
3: Let $e \leftarrow (u, v)$ such that $(u, v) \in p$ and $(u, v)$ has minimum weight
```

```
       $w(u, v)$.
4: $p_a \leftarrow$ shortest-path$(a, u)$; $p_b \leftarrow$ shortest-path$(v, b)$
5: $p' \leftarrow (p_a, u, v, p_b)$
6: $s' \leftarrow (s \setminus p) \cup p'$
7: return $s'$
```

Time complexity: $O(m)$

Improvement heuristic I2 Replaces the weightiest edge from a path in a solution using a shortest-path.

```
1: Select a solution $s$ from memory $M$.
2: $p \leftarrow a \rightsquigarrow b$, for $(a, b) \in s$.
3: Let $e \leftarrow (u, v)$ such that $(u, v) \in p$ and $(u, v)$ has maximum weight
   $w(u, v)$.
4: $p_a \leftarrow$ shortest-path$(u, v)$;
5: $p' \leftarrow (a \rightsquigarrow u, p_a, v \rightsquigarrow b)$
6: $s' \leftarrow (s \setminus p) \cup p'$
7: return $s'$
```

Time complexity: $O(m)$

Improvement heuristic I3 Heuristic based on the “triangle inequality” property.

```
1: Select a solution $s$ from memory $M$.
2: Let $p \leftarrow a \rightsquigarrow b$ be the path in $s$ between $a$ and $b$, for $(a, b) \in S \times D$.
3: for each subpath $x, y, z$ in $p$ do
4:     if $\exists (x, z) \in E(G)$ such that $w(x, z) < w(x, y) + w(y, z)$ then
5:         $p \leftarrow a \rightsquigarrow x, (x, z), z \rightsquigarrow b$
6:     end if
7: end for
8: return $s'$
```

Time complexity: $O(pm)$

Improvement heuristic I4 Creates a new solution from an existing solution, using only shortest paths.

```
1: Select a solution $s$ from memory $M1$.
2: for each source-destination pair $(a, b) \in s$ do
3:     $p \leftarrow$ shortest-path$(a, b)$
4:     Replace $a \rightsquigarrow b$ by $p$ in $s$.
5: end for
6: return $s$
```

Time complexity: $O(pm)$

Destructor D Chooses a solution from shared memory, using a probability distribution criterion:

```
parameter Memory $M_i$
$s \leftarrow$ roullet_weel$(M_i)$
delete(s)
```

Time complexity: $O(|M_i|) = O(k_i)$

5 A partially synchronous distributed model

The A-Teams approach has a remarkable intrinsic parallelism: the application of various heuristics to the same state of the shared memory can be performed concurrently.

In implementing a parallel A-Team strategy, a problem that is faced is that the timing of heuristics is different. This can result in one heuristic being executed many times, while other heuristics have less opportunity to work in the parallel memory due to a greater complexity. This can lead to a situation where, although there is a team of heuristics, only one of them is contributing to the final solution. In the other hand, one can not synchronize the heuristics because this will make all heuristics depend on the slowest of them.

To avoid the timing problem, we use a partially synchronous model to implement the A-Team [11]. In such a model, one can fix a maximum level of asynchrony between tasks in different processors. These tasks are coordinated by an external synchronizer. In this section we define the formal characteristics of this model and describe the partial synchronous implementation of the A-Team.

5.1 The partially synchronous model

We consider a distributed memory system composed by a set of p processors, $p > 1$, each one with its own memory. Like in a computer network, there is no shared global memory, and the processors interact only by message passing through (bi-directional) communication links. This parallel system is represented by an undirected connected graph $H_{ps} = (Q, E(H_{ps}))$, where Q represents the set of processors and $E(H_{ps})$, the communication links. For the remainder of the paper, members of Q are referred to as *nodes*, whereas the members of $E(H_{ps})$ are called *edges*. Every node $q_i \in Q$ is able to communicate directly only with its neighbors in H_{ps}. The set of neighbors of q_i is denoted by $N(i)$.

The timing characteristics of the partially synchronous model are related to the following properties:

(1) Each node $q_i \in Q$ is driven by a local clock, which generates pulses s_i of duration $\tau_i(s_i) \geq 0$. During a pulse s_i, the node q_i can receive a set $MSG_i(s_i)$ of messages from nodes in $N(i)$ (this set is empty if $\tau_i(s_i) = 0$), do some computation locally, and send a set of messages (possibly empty, and certainly empty if $\tau_i(s_i) = 0$) to some nodes in $N(i)$. The computation performed by q_i during s_i changes the state of the local memory of q_i from

$\sigma_i(s_i)$ to $\sigma_i(s_i+1)$. The duration of a pulse is independent on the duration of other pulses, and can be arbitrarily large.

(2) Let msg be a message sent by q_i in pulse s_i to a node $q_j \in N(i)$. If s_j is the pulse in q_j in which msg is received (i.e. $msg \in MSG_j(s_j)$), then $s_j > s_i$ and $s_j \leq s_i + \delta_{ij}$, for some given constant $\delta_{ij} \geq 1$. Thus, the message delay of any communication is unpredictable, but bounded, no matter how long the duration of the pulses of the nodes involved in this communication are. Define $\Delta_i = \max_{q_j \in N(i)} \delta_{ij}$, for all $q_i \in Q$.

5.2 Distributed memory executions

The intrinsic parallelism in an A-Team is better characterized if we consider a parallel model with solutions shared across processors. In such a model, heuristics can execute concurrently and physically share the solutions in its local memory, making requests for solutions in other nodes of the system. In this case, the parallel reading operation returns a solution located at a specified position in M, and such position is selected at random. Still, the parallel writing operation takes a solution as input and must select an entry in M in order to store this solution. The entry is selected using a generalized "roulette wheel" principle.

In our implementation, the memory M is evenly distributed over q processors, with M_i being the portion of the solutions memory assigned to processor q_i. Each heuristic $H_i \in H$ can run in any of the processors and access any of the other $M_j \in M_{h_i}$ memories for reading or writing. The operation `read` and `write` must now take in considerations distributed considerations.

Let $q_i \in Q$ be a processor which requests a distributed writing of a solution s. The first step is the choice of the specific memory M_i that will be written to. This can be made based on many characteristics of memories, or even randomly. The method used considers the *diversification* of each memory, where the diversification of a memory M_i is given by the value of the worst solution minus the value of the best solution in this processor. The choice is made for the memory with least diversification. Our aim is to contribute different solutions such that memories can share potential improvements.

To reduce communication costs, all reading and most of the writing operations are made in the local memory. This decision implies that writing solutions in remote processes is the only communication method between distinct memories. Therefore, we define for each process a parameter α_w, the percentage of solutions that will be written to other (non local) memories.

Every node $q_j \in N(q_i)$ must process requests for writing solutions in its local memory. It must select a position in its local memory independently, and this is done in the same way as in the sequential writing.

5.3 Types of messages

The distributed operations on M involve some message exchange between processes in Q. These messages can be classified as initialization messages, writing messages, and control messages. For the purpose of synchronization, all messages include the pulse σ_t in which the distributed operation was requested to M_j by some heuristic H_i.

5.3.1 Initialization messages

The initialization messages are responsible for proper set up of variables. In the parallel A-Team, the initialization phase is started by a driver process which send a `startup` message to all child processes. The local variables are then initialized. An important sub-step in many of the proposed heuristics is the computation of the shortest path between two nodes. This is a time consuming task, and therefore all shortest paths are computed at the beginning of the A-Teams execution and stored in the matrices Π and A as explained in Section 4.1. To reduce the burden of calculating these values, the all-pairs of shortest paths algorithm is distributed over all processors. The distributed algorithm uses a torus model for diffusion of the computed values [12]. According to this model, values must be computed in two phases (corresponding to the two dimensions of the torus). The values in the first phase are sent using the `send_paths1` message. Then, the second phase of the algorithm proceeds and the result is sent using the `send_paths2` message. After this, the A-Team is initialized and start the processing of messages.

5.3.2 Messages for writing solutions

The second type of messages is used when a process needs to write a solution. As explained, the first step consists in asking the neighbor nodes for a diversification parameter. Based on this parameter the exact memory is chosen. The process then sends a message `ask_parameter` for each node in its neighborhood. Every time this message is received, the diversification parameter is computed and sent together with the `resp_parameter` message. Then, after processing `resp_parameter` from all neighbor nodes, a node can send the solution to the chosen node using the `receive_sol` message. The destination node receives the solution and inserts it in its local memory. Finally, after receiving all data the node sends the message `end_write` to the requesting node. This message acknowledges the correct reception of the solution.

5.3.3 Control messages

The third kind of messages, control messages, are used to define the execution cycle of each process. There are two messages in this group: `work_done` and `terminate`. To understand the necessity of these messages, suppose that the *terminate condition* is true and the algorithm must terminate. How can all processors agree in stopping their operations? More importantly, how can a node be sure that no other node is depending on it for processing messages and therefore it can stop safely? For this kind of coordination, the network must specify a life cycle control protocol.

The simple method employed in our implementation uses one of the nodes (the first one) as the arbitrator for execution termination. When the terminate condition is evaluated as true by the arbitrator (for example, the optimum solution was found), it must send the message `terminate` to all other nodes. In response, the node send the same message as an acknowledgment. Each node that receives the `terminate` message must stop generating messages and only answer to requests previously sent to it. When the arbitrator receives an acknowledgment from all nodes it can safely send the `terminate` message. Finally, in response to the `terminate` message the process can shut down and exit. The processing of messages is summarized in Algorithm 5.3.3. The Algorithm 5.3.3 shows the actions performed in each processor q_i during a pulse s_i.

5.4 Local Variables

Some local variables are used to handle the distributed operations on M. For example, the communication delays in the partially synchronous model allow a process q_i to have a number of pending distributed writings in a pulse. Then, a circular buffer $pending\_write_i$ is used to store, in processor q_i, the solutions associated to a pending writing requested by q_i. The size of this buffer must be equal to δ_i, because this is the maximum difference in pulses between q_i and any other process in the system. The extremes of the circular buffer are indicated by $first\_pending_i$, which is the oldest solution in the circular buffer, and $last\_pending_i$, which is the most recent solution in the circular buffer.

5.5 Simulating the partially synchronous model

In this section, we deal with the problem of implementing the partially synchronous algorithm described in the previous section in an asynchronous model of distributed computation. The asynchrony in this model stems from the fact that all messages are delivered within some finite but unpredictable delay. In

```
**Input:** message msg from node $j$
**switch** msg
{Initialization messages}
**case** startup:
  Compute shortest paths among nodes
    from $(i-1)(m/|Q|)$ to $i(m/|Q|)-1$.
  $j \leftarrow i + m/|Q|$
  Send message send_paths1 to node $q_j$
**case** send_paths1:
  $P \leftarrow$ path received with the message
  copy information in $P$ to $\Pi$ and $A$.
  send $P$ to node $q_{i+1}$
**case** send_paths2:
  $P \leftarrow$ path received with the message
  copy information in $P$ to $\Pi$ and $A$.
{Messages for writing solutions}
**case** ask_parameter:
  $p \leftarrow f(best\_sol) - f(worst\_sol)$
  send message(resp_parameter) with parameter $p$
**case** resp_parameter:
  $resp_j \leftarrow p$, where $p$ is the parameter received
**if** $resp_k \neq \emptyset$ for all $k \neq i, k \in \{1, \ldots, |M|\}$ **then**
    $pos \leftarrow k$ such that $resp_k$ is minimum
    $sol \leftarrow$ random($pending\_sol$)
    $pending\_sol \leftarrow pending\_sol \setminus sol$
    Send message receive_sol to node $pos$, with pa-
    rameter $sol$.
**end if**
**case** receive_sol:
  $sol \leftarrow$ solution encoded in the message
  call write(sol, $M_i$)
  send message end_write to node $j$
**case** end_write:
  $pending\_write \leftarrow pending\_write - 1$
{Control messages}
**case** work_done:
  $stop\_processing \leftarrow$ true
**case** terminate:
  $terminate\_condition \leftarrow$ **true**
**end case**
```

```
1: receive startup message from synchronizer
2: while $terminate\_condition \neq$ true do
3:    receive safe($s$, MSG($s$))
4:    Do message processing (Algorithm 5.3.3)
5:    if $stop\_processing \neq$ false then
6:       $h \leftarrow$ random($H$)
7:       $sol \leftarrow h(M_i)$
8:       $l \leftarrow$ random($|M|$)
9:       if $l = 1$ then
10:         write($sol, M_i$)
11:      else
12:         Send message ask_parameter for all nodes $j$
            such that $j \in \{1, \ldots, |M|\}$ and $j \neq i$
13:         $pending\_sol \leftarrow pending\_sol \cup sol$
14:      end if
15:   end if
16:   send(acknowledge)
17: end while
```

our case, the Parallel Virtual Machine (PVM) running in a local network is the underlying parallel system, therefore delays can turn out to be a major bottleneck. The program must solve two main issues, namely keep the synchronization delay within the specified limits (δ_i) and implement a fair distribution of processing across nodes in the system.

A simple way to solve this problem is to have, for each processing node of the A-Team, an auxiliary process which takes control of the communication requirements. Figure 3 represents graphically this architecture. Each node is now composed of two sub-tasks: the A-Team and a synchronizer module. All communication from and to the A-Team task is handled by the synchronizer. In the other hand, messages can be sent directly among synchronizers running in different nodes. This ensures that the messages are exchanged within the specified delays.

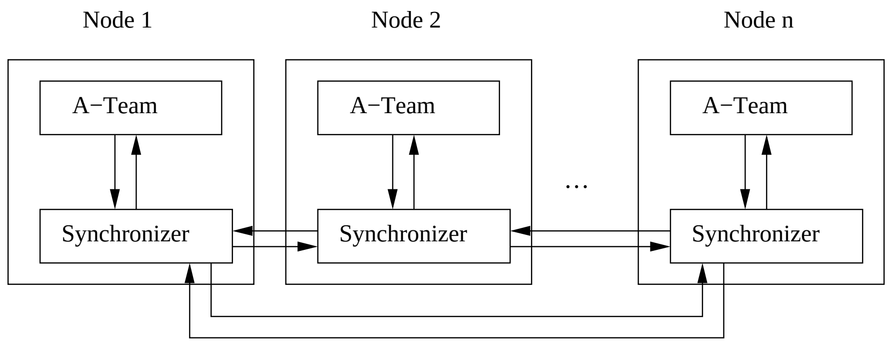

Fig. 3. Organization of processes in the nodes of the parallel machine.

In this architecture, only a specified number of messages can be exchanged between the A-Team and its synchronizer. The following types of messages are used:

- *startup*: this message signals the start of the computation. In response to this message all initialization tasks are executed. The most important of these tasks is the distributed computation of the matrices $A_{n\times n}$ and $\Pi_{n\times n}$, with all pairs of shortest paths, and the matrix of precedents in the shortest paths, respectivelly.
- $safe$: sent by the synchronizer when it is safe to do new computations. This is an important message because all work of the A-Team is done as a response to $safe$.
- *acknowledge*: sent by the A-Team process to acknowledge the receiving and processing of the safe message.

In addition to these types of messages, there is a specific way for the A-Team to send information to other nodes. In order to do this, the task must send a message to the synchronizer with an special tag, specifying the destination. This tag is just a number between 1 and $|Q|$. The A-Team task does not need to know the exact identifier of the other nodes, thus simplifying message processing. All messages received from other nodes are also handled by the synchronizer. The A-Team just needs to recover the node tag from each message. Each message (sent or received) must also include the current execution step s_i.

6 Computational Results

In order to evaluate the proposed algorithm we executed two types of experiments. The first kind of experiment used a sequential implementation (number of nodes equal to one). The second type of experiment tested the speedup due to the parallel implementation.

6.1 Hardware and software configuration

The parallel A-Team was implemented using the PVM (Parallel Virtual Memory System) [13]. The programming language used was the C language, with the GNU GCC compiler. The machine used was an IBM SP2 parallel system, with processors running at 66.7MHz and 256MB of memory in each node.

In the tests we used random instances presented in [14] to evaluate and compare the performance of our method.

The total size of memory for solutions used in the A-Teams is equal to the number of nodes n. During tests this value showed to give good results for graphs with the size that we are considering. The value of δ was set to 4 for each node q_i, according to some experimentation performed on the data.

6.2 Sequential tests

The first experiment executed tried to assess the computational performance of the A-Team approach, compared to other methods. We did two kinds of comparisons: against both an exact method and an approximate algorithm. The exact method chosen was a branch-and-cut, implemented by Meneses [15]. Due to the hardness of the problem, the branch-and-cut algorithm could find exact results only for graphs with at most 40 nodes, 500 edges and $p = 5$. For most instances the A-Team found the optimum value in a fraction of the time spent by the exact algorithm. In Tables 6.2 and 6.2 we have a summary of results for the first experiment. The last columun in this table represents the distance between the average solution found by the A-Team and the optimum value, when these values are different.

The second experiment was executed for instances with sizes ranging from 100 to 200 nodes, 500 to 2000 edges and $p = 10$ to 20. For these graphs we tried to solve the problem using both the A-Team and a Down Hill greedy heuristic. The results of these experiments appear in Table 3. We can see that the improvements in solution value are between 29% and 38%. The greater running time of the A-Teams can be explained by the fact that it needs to initialize the memory, which increases with the size of the instance, something that is not required in the local search algorithm.

6.3 Parallel tests

The idea of the parallel tests is: given the same A-Team configuration as in the sequential tests, what speed up can we have with the increase of nodes in the parallel system? With this goal in mind we run the program for instances shown in Table 4.

The organization of the parallel processes was done as follows. Initially, a parent process in launched in the first node of the SP2 machine. The responsibility of this process is just to launch identical copies of the `ateam` program in each of the remaining nodes. The parameters passed to the child processes are the instance to be solved and the random seed used throughout the computation. Thereafter, each node starts executing the partial synchronizer, that controls the computation. The organization of processes in each node is described in Figure 3.

Instance			**Branch-and-Cut**		**ATeam**				
n	m	p	Opt. sol.	CPU time	Best run	Suc. ratio	Ave. sol.	Average time	Dist. (%)
30	60	8	57	28s	57	10	57.00	32s	
30	60	7	41	14s	41	10	41.00	39s	
30	60	6	38	15s	38	10	38.00	12s	
30	60	5	39	24s	39	10	39.00	05m08s	
30	100	8	29	01m51s	29	1	29.80	05m02s	2.76
30	100	7	25	09s	25	10	25.00	02m03s	
30	100	6	30	49s	30	10	30.00	01m04s	
30	100	5	21	01m14s	22	0	24.70	03m33s	17.62
30	200	8	21	02m50s	21	10	21.00	03m28s	
30	200	7	19	52m55s	21	0	21.50	15m24s	13.16
30	200	6	18	17s	18	10	18.00	23s	
30	200	5	22	42s	22	10	22.00	06s	
30	300	8	13	19m31s	13	8	13.20	18m46s	1.54
30	300	7	13	02m59s	13	10	13.00	01m46s	
30	300	6	11	04m14s	11	10	11.00	23s	
30	300	5	11	01m36s	11	10	11.00	25s	
30	400	8	14	48s	14	10	14.00	09m36s	
30	400	7	14	59m43s	14	10	14.00	02m28s	
30	400	6	12	12m42s	12	10	12.00	02m30s	
30	400	5	10	01m19s	10	10	10.00	14s	

Table 1
Computational results comparing A-Team to a branch-and-cut algorithm for instances with 30 nodes.

The results obtained are summarized in Table 4. The last columns represent the average time required to find the same solutions of Table 3. The columns labeled S represent the speedup of the corresponding configuration. The speedup is defined as

$$S(n) = \frac{T(1)}{T(n)}$$

Instance			**Branch-and-Cut**		**ATeam**				
n	m	p	Opt. sol.	CPU time	Best run	Suc. ratio	Ave. sol.	Average time	Dist. (%)
40	100	8	50	02m08s	50	4	50.70	14m18s	1.40
40	100	7	48	01m37s	49	0	49.00	10m08s	2.08
40	100	6	38	25s	38	10	38.00	01m05s	
40	100	5	32	47s	32	10	32.00	11s	
40	200	8	30	08m24s	30	1	31.10	14m24s	3.67
40	200	7	19	32s	19	10	19.00	1m26s	
40	200	6	23	59s	23	4	23.60	08m22s	2.61
40	200	5	16	07s	16	10	16.00	58s	
40	300	8	20	01m48s	20	2	21.00	10m31s	5.00
40	300	7	14	01m19s	14	10	14.00	07m01s	
40	300	6	14	01m27s	14	10	14.00	56s	
40	300	5	20	06m27s	20	10	20.00	01m15s	
40	400	8	22	39m35s	22	5	22.70	09m30s	3.18
40	400	7	14	08m37s	14	10	14.00	48s	
40	400	6	17	02m17s	17	10	17.00	01m36s	
40	400	5	11	01m42s	11	10	11.00	38s	
40	500	8	17	04m07s	17	10	17.00	07m29s	
40	500	7	16	04m33s	16	10	16.00	01m14s	
40	500	6	14	03m46s	14	10	14.00	01m04s	
40	500	5	12	10m45s	12	10	12.00	01m16s	

Table 2
Computational results comparing A-Team to a branch-and-cut algorithm for instances with 40 nodes.

where $T(1)$ is the time for the sequential program and $T(n)$ is the time for the parallel program with n nodes. We can see that with the increase of the number of nodes, the speed up of the system also increases. This is due to a greater number of possible solutions which can be simultaneously generated, but there are also reductions in the time for initialization and for finding the solutions to the shortest path problem.

Instance			Local Search			A-Team			
			best	average	aver.	best	average	aver.	improv.
n	m	p	sol.	sol.	time	sol.	sol.	time	
100	500	10	77	93.2	1.5	59	63.3	8.8	32%
100	600	10	56	69.4	1.5	42	43.6	10.5	37%
100	700	10	66	74.2	1.2	43	46.8	11.3	37%
120	800	12	80	87.4	2.4	50	54.1	16.9	38%
120	900	12	73	91.3	2.1	60	64.3	17.5	30%
120	1000	12	56	63.6	2.2	41	43.1	18.5	32%
140	1000	14	82	98.2	3.1	61	65.4	29.6	33%
140	1100	14	65	81.1	3.8	56	57.5	31.7	29%
140	1200	14	65	74.9	3.3	47	50.6	31.9	32%
160	1300	16	108	121.6	4.8	78	80.7	54.9	34%
160	1400	16	97	112.3	4.8	70	77.7	59.6	31%
160	1500	16	89	97.1	4.9	55	64.2	60	34%
180	1600	18	101	115.3	6.4	73	80.3	93.7	30%
180	1700	18	113	122	6.5	75	79.4	93.2	35%
180	1800	18	99	110	6.5	69	73.4	98.4	33%
200	1800	20	119	133.6	8.7	87	91.4	131.3	32%
200	1900	20	114	126	8.7	79	84.6	133.7	33%
200	2000	20	112	122.5	8.6	77	83.0	151.2	32%

Table 3
Computational results comparing A-Team to a local search algorithm.

7 Concluding remarks

In this paper we discussed the Point-to-point Connection Problem, a NP-Hard problem with applications in the areas of telecommunications and VLSI. We presented a parallel, randomized optimization algorithm for the PPCP and discussed how it can be implemented using a partially synchronous model.

Experimentation with the algorithm shows that it gives good results compared with an exact method (branch-and-cut) and another randomized technique (a down-hill algorithm). The experiments also suggest that the performance of the parallel algorithm improves as the number of nodes in the system increases.

Instance			**Average time per number of nodes**								
n	m	p	1	2	S	4	S	8	S	16	S
100	500	10	8.8	6.00	1.47	5.00	1.76	3.66	2.41	2.43	3.62
100	600	10	10.5	7.50	1.40	6.00	1.75	4.40	2.39	2.91	3.60
100	700	10	11.3	9.00	1.26	6.00	1.88	4.62	2.44	3.12	3.63
120	800	12	16.9	12.30	1.37	9.00	1.88	6.96	2.43	4.57	3.70
120	900	12	17.5	12.48	1.40	9.00	1.94	7.34	2.38	4.78	3.66
120	1000	12	18.5	12.50	1.48	9.98	1.85	7.74	2.39	5.01	3.69
140	1000	14	29.6	18.54	1.60	15.19	1.95	12.13	2.44	8.05	3.68
140	1100	14	31.7	20.98	1.51	16.91	1.87	13.17	2.41	8.76	3.62
140	1200	14	31.9	23.08	1.38	17.27	1.85	13.72	2.33	8.59	3.71
160	1300	16	54.9	36.27	1.51	28.98	1.89	22.91	2.40	14.93	3.68
160	1400	16	59.6	41.25	1.44	32.08	1.86	24.64	2.42	16.45	3.62
160	1500	16	60	39.80	1.51	32.24	1.86	25.21	2.38	16.36	3.67
180	1600	18	93.7	67.78	1.38	50.18	1.87	39.77	2.36	26.00	3.60
180	1700	18	93.2	65.96	1.41	47.98	1.94	40.05	2.33	24.79	3.76
180	1800	18	98.4	67.98	1.45	53.03	1.86	41.78	2.35	26.17	3.76
200	1800	20	131.3	91.37	1.44	69.06	1.90	53.84	2.44	35.52	3.70
200	1900	20	133.7	92.08	1.45	73.04	1.83	56.44	2.37	35.79	3.74
200	2000	20	151.2	100.03	1.51	81.22	1.86	63.55	2.38	41.69	3.63

Table 4
Computational results comparing the speed up of a parallel A-Team with 2, 4, 8 and 16 nodes.

The PPC problem is still not well studied and there are many opportunities for improving the solutions to this problem. The techniques presented in this paper are also not yet in widespread use, but can be successfully applied in many other combinatorial optimization problems.

8 Acknowledgments

Thanks to the CENAPAD-NE center (Centro Nacional de Processamento de Alto Desempenho do Nordeste, Brazil) for the parallel environment used in the implementation of the software, and Paulo Benicio (UFC) for helping with the PVM system.

References

[1] J. Kurose, A. K. Ross, Computer Networking: A Top-Down Approach Featuring the Internet, Addison Wesley, 1999.

[2] C. Li, S. McCormick, D. Simchi-Levi, The point-to-point delivery and connection problems: complexity and algorithms, Discrete Applied Math. 36 (1992) 267–292.

[3] M. Natu, S. Fang, Network loading and connection problems, Tech. rep., North Carolina State University (1995).

[4] M. Goemans, D. Williamson, General approximation technique for constrained forest problems, SIAM J. Comp. 24.

[5] P. Souza, S. Talukdar, Asynchronous organizations for multi algorithm problems, in: ACM symposium on applied computing, Indianapolis, 1993.

[6] H. Longo, Solving the set covering and partition problems using A-Teams, Master's thesis, UNICAMP (State University of Campinas, Brazil), (In portuguese) (1995).

[7] V. Cavalcante, P. Souza, Solving the job shop schedule problem by asynchronous teams, in: Proc. of Int. Symposium on Operations Research with Applications in Engineering, Technology and Management (ISORA'95), 1995.

[8] S. Chen, N. S. S.N. Talukdar, Job-shop-schedule by an A-Team of asynchronous agents, in: IJCAI-93 Workshop on Knowledge-Based Production, Scheduling and Control, Chambery, France, 1993.

[9] S. R. Gorti, S. Humair, R. D. Sriram, S. Talukdar, S. Murthy, Solving constraint satisfaction problems using a-teams, AI-EDAM 10 (1996) 1–19.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.

[11] R. C. Correa, A synchronizer for partially synchronous parallel computations, Tech. rep., Universidade Federal do Ceará, Brazil (2001).

[12] V. C. Barbosa, An Introduction to Distributed Algorithms, MIT Press, 1996.

[13] V. S. Sunderam, PVM: A framework for parallel distributed computing, Concurrency: practice and experience 2 (4) (1990) 315–339.

[14] F. C. Gomes, A. G. Lima, C. A. S. Oliveira, C. N. Meneses, Asynchronous organizations for solving the point-to-point connection problem, in: Proceedings of International Conference on Multi-Agent Systems (ICMAS), IEEE Press, Paris, France, 1998.

[15] C. Meneses, Private communication, research on UNICAMP (State University of Campinas, Brazil) (2001).