協調探索における通信戦略の研究

楢崎 修二

https://doi.org/10.11501/3111011

出版情報：Kyushu University, 1995, 博士（工学）, 論文博士
バージョン：published
権利関係：
Chapter 4

Evaluation and Discussion

In this chapter, we evaluate two strategies through simulations using Traveling Salesman Problem (TSP) as an example of a search problem. Under varying communication costs, both strategies show good performances. We also discuss the adaptability and extensibility of our strategies.

4.1 Traveling Salesman Problem as a Cooperative Search

We measure the quality of the strategies through simulations using the Traveling Salesman Problem (TSP). TSP was defined by A. J. Hoffman and P. Wolfe in p. 2 of [LLARKS85] as:

The TSP for a graph with specified edge lengths is the problem of finding a Hamiltonian cycle of shortest length.

A Hamiltonian cycle is a cycle that contains all the vertices of the graph exactly once. TSP is a well-known NP-hard problem.

We implemented the range control strategies on both flat spatial structures and hierarchical ones, the frequency control strategy and a fixed strategy for comparison. And search algorithm we used is the *branch-and-bound* method on search agents that run in parallel. They exchange the cost of the current best path as a threshold value. Since the quality of the threshold increases monotonously, merging some pieces of information (threshold) means only selecting the best one among them. It relieves the expectation cost. The pseudo-coded algorithm is the following:
put initial state in global bag
do in parallel {
    while (global bag is not empty) {
        pick up a node from global bag that is better than local threshold
        expand it and return new nodes to global bag
        if (a new node is better than threshold)
            update local threshold and multicast it
    }
    update history
}
report the best path

Italic statements in the code is for cooperation.

In this simulation, the number of cities is 10 and the length of histories is 10. The system consists of 100 agents. In this case, the threshold updates over 200 times. After each expanding node, the difference between the value of current threshold and the one before the expansion is stored in the history memory of an agent. Since the history holds the revision of threshold, we can calculate the expected values of threshold that other agents get most recently. Thus a Monte Carlo simulation on the history gives the agent the expected best value among $n$
4.2 Results of Simulations

Our simulator was build on Common Lisp. Exactly speaking, we use CLOS/MOP's features. Each agent is implemented as an instance of a class. Its details will be described in the next chapter. Each agent runs in round robin. Dispatch granularity is the same as a main method. Communications in a stage occur simultaneously at the end of the stage. When an agent resumes, it checks its queue for received packets. Thus all messages are received the next stage of the sending stage. Every range control strategies can change the communication range (cluster size) in ±2 at each step. They evaluate the utility of communication using the five points: not change, increment by one or two, and decrement by one or two. The strategy selects best one out of them. Though this process can not select the best size at once, this implementation avoids oscillation. Figure 4.4 shows a sample of the changes of the cluster size. We fixed the optimal size to 10. But at the first step, all cluster sizes are one; each agent belongs to a cluster. They must change their cluster sizes to the optimal ones. Though some clusters achieve the optimal size quickly, in general, our structure-changing protocol that is mentioned at the previous chapter requires a long time for this transition. We expect the changes of the optimal size in real problem solving is small in a short period.

Through all experiments, we ignore the cost of getting subproblems from the global bag.
The number of cities is 8. Thus the number of possible solutions is $8! = 40320$.

4.2.1 spatial connectivity control strategy

First, we examine the frequency control strategy. As we described in the previous chapter, there are three substrategies:
4.2. Results of Simulations

Figure 4.4: changes of clusters' size

- fixed in flat structures
- fixed in clustered structures
- changing spatial structures during execution

Here we examine all three strategies. The first two strategies only change the number of receivers. Last one has a threshold that is the trigger to change the spatial structure. In this examinations, we give the threshold a priori. If an expected communication range is smaller than the threshold, agents apt to use flat structures. Otherwise, agents attempt to use cluster structures. Exactly speaking, in order to avoid a perturbation between two structures, our implementation use $\alpha \times \text{threshold}$ for the trigger to transit from flat to clustered, and $1/\alpha \times \text{threshold}$ for the reverse transition. We use 4.0 as the threshold and 0.81 as $\alpha$.

Results are shown in Figure 4.5,4.8, and 4.9. They show that the strategies achieve good performance against affected communication costs. The data are averages of 10 ~ 100 experiments.

Figure ?? and 4.7 compare the two estimation methods. Formula 3.6 could achieve as good performances as the Monte Carlo method in TSP.
The communication cost function in Figure 4.5 are simple ones. The ones used in Figure 4.8 non-linear functions. This non-linearity is introduced to imitate the congestion on network. In Figure 4.9 that shows the traces of the size of the range with the communication cost at Figure 4.8 (a), the solid line shows the range control strategy on flat structures, the dashed line shows the optimal size on hierarchical structures, and the dotted line is the real size of clusters on hierarchical structures. Strictly speaking, the communication cost used in Figure 4.9 (b) is different from the one in Figure 4.9 (a). The cost function hardly affects the tendency of the range change.
4.2. Results of Simulations

(A) communication cost/computation cost = 0.01x

(B) 0.001x

(C) 0.0001x

(D) 0.1 log(x)

(E) e^{0.0001(x-1)} - 1

Figure 4.5: results of spatial strategies(1)

The average time of a single agent system is 21501. Thus the region under 215 means superlinear.
Evaluation and Discussion

Figure 4.6: difference of Monte Carlo simulation and $n/(n+1)$ estimation
communication cost/computation cost is $0.0001 \times$
4.2. Results of Simulations

Figure 4.7: Difference of Monte Carlo simulation and $n/(n+1)$ estimation(2)

communication cost/computation cost is 0.01x
Evaluation and Discussion

Figure 4.8: results of spatial strategies (2)

Figure 4.9: results of spatial strategies (3)
4.2. Results of Simulations

Figure 4.10: traces of the size of the range
4.2.2 frequency control strategy

The second category of strategies is the frequency control strategy. Our current implementation is based on the manager-worker model and different from a structure for the range of costs.
4.2. Results of Simulations

Figure 4.12: results of frequency strategy(2)

control strategy in which agents are identical in the initial state. Only one manager determines the frequency of multicast based on the result of its local computation. If it decides that it is a time to exchange local knowledge, it multicasts a request for exchanging information. Receivers send back their knowledge to the manager. Then the manager multicasts the best knowledge of the received pieces. In this three-phase communication, every agent should stop local computation.

Most parameters for experiments for the frequency control strategy is the same as the ones
in the experiments for spatial connectivity control strategy. The number of agents is 100: one manager and 99 workers. Figure 4.11, 4.12 is the results.

4.2.3 results

The properties we found are summarized below:

1. Changing the range and the frequency during the execution is useful. The initial range hardly affects the performance in both flat and hierarchical structures. As Figure 4.10 shows, there are three stages from the view of the size of ranges on both structures. In the first stage (at 0 ~ 20 step) and the last stage (over about 150 step) the communication range is relatively small. This result is derived from the nature of the searching process. At the initial stage, for no agent finds a new threshold, the strategy reduces the communication range. In the middle stage (at 20 ~ 150 step) where the distribution of agents' information becomes large, the range becomes very large. In particular, the strategy on hierarchical structures changes sharply, because an information collector of a cluster controls the size of the cluster. The best answer was often found at 80 ~ 150 step. At the last stage after finding and broadcasting the best answer, updating the threshold never occurs and no more communication is needed. In conclusion, the more frequently information is renewed, the more frequently or more widely identical agents communicate. But excessively frequent updates decrease the number of communication.

2. In the frequency control strategy, the frequency of communication is affected by the communication (broadcast) cost. Similarly, the size of communication range decreases if its cost is high where the utility of communication becomes low.

3. There is no clear difference between flat structures and hierarchical structures. The reason is the peak size of ranges is not so large in this simulation and changing cluster size requires some other communication on hierarchical strategies. This reduces the merit of hierarchical structures. However, experiments with other cost functions showed large communication costs make the difference clear. For example, in Figure 4.5 (A), (B) and (C), the difference becomes small in its order. Generally, the structure changing strategy is better than the strategy on cluster structures.
4.3. Discussion

4. A communication range becomes larger after the difference between the best knowledge and the worst knowledge in time becomes large. A history has enough sensitivity. This result corresponds to the result that we mentioned in the previous chapter.

5. The best threshold spreads quickly when it is found. Sometimes search terminates before every agent knows it. But usually most agents know it. Since, compared to flat structures, cluster structures apt to make closed groups in each clusters, expansion speed tends to be slower than the one on flat structures.

The properties described above are held in other communication cost functions from \( O(\log(n)) \) to \( O(\exp(n)) \). They hardly depend on a cost function if it increases monotonously. In conclusion, programmers need not care about the communication topology by using the strategies.

4.3 Discussion

4.3.1 applicability of the strategies

In this section, we discuss the applicability of these strategies to other fields and ways of implementing these strategies. Related works are also described.

another search algorithms

First, we think more about the branch-and-bound method. As we described, our implementation ignores the communication cost for gathering subproblems into a blackboard. We justify this assumption as we can implement it as distributed memory easily. But if so, we should deal with the local starvation of the problem. Of course, any agent can multicast or send a request to neighbors when the local bag is empty. But agent must wait for its results. If we do not want to make agents idle, agents sent a request of subproblems to other agents before it becomes really idle. Thus we had better build a cooperative communication strategy to require subproblems. A local history about the consumption rate will be a help. Then we can use our strategy, where it control the flow of subproblems instead of a threshold. Furthermore, by using a history of requests from other agents, it would behave more cooperatively.
Second topic is on A* search. The current shortest path as a threshold in TSP is a kind of information that is acquired during execution. The search algorithm used here does not use any estimation heuristics. In a general case, we can use an estimated value as the measure for modeling execution status. Considering A* search [Peh84], the quality of a node (subproblem) is measured by an evaluation function. Thus we can use the strategies for exchanging subproblems with the relations in Table 4.1.

Another cooperative searching method is bidirectional search [Peh71, Ish93b]. It is search from an initial state and a goal state simultaneously. Two search processes look for a path that they can meet. Locations of each agent in the search space will help for cooperation. Thus by giving a heuristic function like the proceeding rate, two agents behave in a cooperative manner; for example, agent require less computational time when it has not proceeded. A history will provide a basis for cooperation. Here, the cooperative strategy is not for communication. It changes an agent's behavior itself. But if the communication cost is very high, the communication strategy that can forecast other agents' states will be useful again.

**out of search problem**

Though we model only cooperative search problem, the connectivity changing strategies are applicable to a lot of areas. First we concern with the *contract net protocol* [DS88]. Contract net protocol is a method for distributing subproblems to appropriate agents. Each assignment takes three phase communication among agents; issuing a task announcement, submitting a bid, and making an award. A merit of this method is fairness of the result. These message are broadcasted. But with a history of replies, agents omit overspreading of messages. Agents use multicast instead of broadcast.

*shared object management* Second example is shared object management. Exchanging threshold can be thought of as algorithms for managing the consistency of threshold that
is shared \textit{weakly} among agents. In addition, the idea of using a local history as a model of an environment can manage shared objects with strong consistency. Usually, strongly consistent objects are used much more than weak consistent objects on distributed systems. Thus the programmers’ burden of keeping coherency effectively will be relieved in a number of application areas.

In this case, the measure of execution is the ratio of local access and remote access. Values exchanged among processors are shared objects themselves. If the current cost for remote access is higher than the expected cost for managing coherency, a shared object should be duplicated and distributed. On the other hand, if it is low, the copies should be merged to reduce the cost in writing. If agents (copies of a shared object) know the access ratio sufficiently with the access history, the strategies would work well.

\subsection*{4.3.2 dealing with heterogeneity}

Our approach assumes the homogeneity of agents. However, the history-based expectation mechanism could apply to not only homogeneous environments but also heterogeneous ones. The history of the revision of information at other agents in current implementations is not separated from its own history. Agents exchange the information with each other and update their histories with the received information. But in heterogeneous network, this dispersion becomes an important problem. An implementation and the evaluation of strategies based on separated histories are of a future plan. Such an implementation will also work on heterogeneous agents.

Another important heterogeneity is in decision making; diversity of actions of agents. If heuristics can find only the most promised search direction, the best path in time does not always lead to a way to the best answer. Thus for reducing the risk, diversity of search directions in some degrees is required. Lesser called such a search strategy that includes this diversity control as cooperative control [Leg91]. We think temporal diversity of the selection of the next problems can be acquired from probabilistic way based on the local history, that makes an appropriate distribution of all agents in the system in a search space. But further study is required for this direction.
Last heterogeneity is about agents’ goals. Since we focus on DPS in which the goal is shared among all agents, situation like Prisoner’s dilemma [Axelrod 1984] do not take place. But a kind of fluctuation or oscillation of decision making may take place. This problem is discussed in [KHH89]. Since our methods, however, use histories, we do not think drastic loss of the performance happens that is reported [KHH89].

Finally we summarize the limitations of the proposed scheme again:

1. Communication costs must be known before execution. It would be difficult on a system that has a hierarchical topology or a large-scale open system.
2. The strategies assume homogeneity of agents. But we gave some possible solutions to it above.

4.4 Summary

- Through simulations of TSP, we have evaluated the quality of our strategies. They showed better results than fixed communication systems for a wide range of communication costs.
- A history is a good model of execution status. There is no clear difference between the monte Carlo simulation and formula 3.6. They estimate the probability of information update well in the range control strategy.
- We discussed problems to apply our strategies to other search algorithms or other application domains. By proposing extension plans, we claimed that many of them will be solved.
- We also discussed some assumptions to use our strategies: the heterogeneity of agents and knowledge on cost functions.
Chapter 5

Separate Description of Communication Strategies

5.1 Communication Strategies as Metalevel Computation

Our communication strategies change the behavior of an agent by using a history that is a measure almost independent of a problem-level program. They are invoked when some changes in agent local status are happen. And after finishing their job, the control flow returns to the invoking point in the problem-level program. This flow reminds us a usual subroutine calls. But since when to invoke communication strategies is defined in the context of the strategies themselves, the interpreter can know it. This means programmers need not to write procedure calls down in their problem-level programs. In other words, communication strategies can be embedded into the semantics of reference and update of local variables in problem solving programs. It is a metalevel computation.

Thus we think deciding communication structures should be a kind of computational reflection [MN88]. This means communication strategies can be separated from problem-level programs and be a metalevel description of program solving agents.

This separation approach has two merits:

1. Programming is divided into two independent subtasks: building communication strategies and writing problem-solving codes. As a result, writing codes for communication
strategies can be omitted from problem-solving programmers’ task. It also makes reusing the communication strategies easy, if there is a generalized protocol between agent and strategy. Communication strategies can be stored in a library. In Werner’s words, we can distinguish “system programmers” and “application programmers” with this framework [Wer92]. As a result, communication strategies and application programs can be developed and tested independently.

2. Since the communication strategies are included in the semantics of a language, the strategies are invoked automatically when needed. For example, if a communication strategy is defined as metalevel computation of the assignment to a slot that holds information to exchange, every modification of the slot invokes the strategy. We can omit explicit invocation codes from problem-level program.

We made such a system on an object-oriented language. We explain this in the following sections.
5.2. Implementation on CLOS Meta-Object Protocol

5.1.1 related works

Relation between agent programming and object-oriented programming languages have been investigated by a lot of researchers [NT90, Hew77, MIT90, YSTH87, MWY91, Gas92b, FB88, FC91, EPT94]. Most agent-oriented languages are based on first-order modal logic, since logic-based representation of specification can infer the agents' mental states like belief, desire and intention in a uniform manner, which select the agent behaviors. On the other hand, concurrent object-oriented languages with extensions for cooperation will be useful especially for DPS. Since our focus on language is how to separate problems and strategies, we do not intend to build a new language but build a programming style. But here we give a short summary on relation between DPS and OOPL.

[MWY91] is an approach to change behaviors in a group with metalevel computation. In this system a group has a metalevel. This is different from classical meta-object languages in which an object has a meta-object. Consistency in the group can be held easily. On the other hand, it requires frequent communication between nodes. Thus we do not think that this approach is rational in cooperative system on distributed systems.

Some concurrent reflective OOPLs like [FB88, FC91] have been used for DPS. For example, Ferver's Mering IV [FC91] is an example of object-oriented languages with meta-level computation for cooperative computation. His focus is providing high level communication primitives to programmers and not providing adaptive cooperative methods based on local computation. But there was no clear separation between cooperative strategy and user application program.

Some Object-oriented OS uses reflection for a method of load-balancing and process migration [OIT93, Yok93]. But the facility is embedded in OS and can not use the semantics of application program well. We think more strongly combination between strategy and application is required. Compiler's support is crucial.

5.2 Implementation on CLOS Meta-Object Protocol

Some object-oriented-programming languages have meta objects that represent behaviors of objects. In particular, CLOS (Common Lisp Object System) [S+90, Kee89] has flexible interface to meta level programming, which is known as CLOS MOP (Meta-Object Protocol).
74 Separate Description of Communication Strategies

[KdRB91, Pae93b]. Thus we decided to build a DAI platform with the strategies on CLOS.

In the next section, we give a short description of CLOS/MOP.

5.2.1 meta-object protocol

A merit of object oriented programming languages (OOPL) is modularity of programming components or objects. A definition of a data structure and procedures for it are encapsulated on a description block called a class. An instance of a class behaves as defined in its class. Some OOPLs can control behaviors of a class itself, that includes ways to make an instance, delete an instance, control method dispatch, inherit super classes and so on. These extensions are done by making class itself an instance of a class: a metaclass. In such languages, all classes have its metaclass. By letting all metaclasses accept same methods (called protocols), we can change the behaviors of instances of a class easily. We can select most appropriate metaclass for problem solving. Since such an extension to a user defined class is not modification of an existing system but an addition of new behaviors for a particular class, it is safe and easy for programmers. A group of researchers had proposed a protocol for the CLOS class structure. It became a standard known as MetaObject Protocol (MOP). CLOS MOP provides a way to change the behavior to access to a slot, update of it, make a new instance of the class and so on. For example, MOP has been used to emulate other inheritance/message-passing models (Dvorak's hybrid knowledge representation tool [DB93] and CLOVERS(ftp://swiss-ftp.ai.mit.edu/archive/clovers/clovers-design-notes.text)) and to build a persistent OOPL system (for example, PCLOS [Pae93b], AllegroStore(a commercial product based on Alegro Common Lisp), ITASCA ODBMS (ITASCS Systems Inc.). Lisp FAQ (http://www.cs.cmu.edu:8001/Web/Groups/AI/html/faq/lisp/top.html) summarizes more implementations).

Now the following is a summary of requirements for separate description of our communication strategies.

1. Change the behaviors of access/update of a particular slot of an agent. Our strategies store the history of renewal of a slot. The strategies that also include a procedure of update the history should be invoked whenever the slot is updated. Or some strategies
5.2. Implementation on CLOS Meta-Object Protocol

should be invoked when the slot is accessed. This is a kind of extension of the semantics of the language.

2. The invocation should be transparent for programmers. Programmers who write problem-solving programs should never concern about communication strategies as much as possible. They do not want to call any procedures for a strategy explicitly.

3. Communication strategies can send and receive some sorts of messages. The messages should be invisible from problem-level programs. And a particular message dispatcher can be invoked when a message is received. This dispatcher must coexist with the dispatcher in problem-level program.

CLOS MOP provide features to meet these requirements. Thus we decide to implement the communication strategies with MOP. In the following sections, we explain our implementation.

5.2.2 class structure

Figure 5.2 shows the class hierarchy in CLOS. It also includes our new classes for describing communication strategies. In CLOS, every object is either one in CLOS class hierarchy or one of a built-in type. all objects, even if they belong to a built-in data type, are subclasses of the object T. T's superclass is T itself. top class of CLOS class hierarchy is standard-object. It defines the primitive behaviors of all instances. Both of slots and methods themselves are subclasses of standard-class.

agent-class and agent-meta-class

First, we should define a new meta class agent-meta-class for a specifier of class behaviors. Its definition is as simple as listed below. Since slots holding information for communication strategies are stored in an instance of each strategy's class, agent-meta-class has no own slot. It is used only for slot instantiation. With it, as a base class of users' agent class, we define agent-class.

(defclass agent-meta-class ()
  ()

1In the Lisp dialects, T means true usually.
Separate Description of Communication Strategies

Figure 5.2: class hierarchy

```
(defclass agent-class () :super class is T.
  ((receive-counter :accessor receive-counter :initform 0)
   (send-counter :accessor send-counter :initform 0)
   (agent-id :accessor agent-id :initarg agent-id))
  (:metaclass agent-meta-class))
```

agent-class has two methods for inter-agent communication. They update the above two slots in agent-class.

```
(defmethod send ((self agent-class) (receiver/s object) ...)
  (defmethod receive ((self agent-class) (message t)) ...)
```
Our strategies use a history of slot renewal. It is a model of program execution. Since we assume that a history of the value renewal of a slot is used as the measure of program execution, the slot object must hold the history of itself. Thus it is rational to use an instance as a wrapper to hold them at the same time. All strategies have a common interface (protocol): a procedure in accessing the value, one in updating the value, and one in receiving messages from another agent. Thus we make each strategy a class. An abstract class: \textit{communication-strategy} is one of their superclasses. It can be defined as follows.

```lisp
(defclass communication-strategy ()
  ((value) ; the wrapped value itself
    (model) ; a history of value
    (communication-cost) ; a function to estimate communication cost
  ))

(defgeneric ref-strategy ((class communication-strategy) instance slot)
  (defmethod ref-strategy ((class communication-strategy) instance slot) nil)

(defgeneric set-strategy ((class communication-strategy) instance slot)
  (defmethod set-strategy ((class communication-strategy) instance slot) nil)

(defgeneric dispatcher ((class communication-strategy) instance slot)
  (defvar *out-of-local-computation-p* nil)
  (defmethod dispatcher around ((class communication-strategy) instance slot)
    (let ((*out-of-local-computation-p* t))
      (call-next-method)))
)
```

Note that method \textit{ref-strategy} is required to update the history of local computation that is stored in \textit{model} slot. \textit{*out-of-local-computation-p*} is a flag variable that is used to distinguish the value from another agent from one found locally.

\textbf{specify metaclass-designator}
All classes must have a metaclass that is either the same metaclass of their superclasses or a valid metaclass. The generic function: `validate-superclass` checks its validity whenever a new class is defined. Thus we must add a method for our `agent-metalevel-class`.

```
(defmethod pcl:validate-superclass
  ((class agent-metalevel-class) (superclass pcl:standard-class))
  t)
```

With this method, we can define any agent classes as subclass of `standard-class`. They use `cooperation-effective-slot-definition` for specific slots.
5.2. Implementation on CLOS Meta-Object Protocol

add slot-access-using-class method

In CLOS, the most primitive access/assign functions are slot-value-using-class and (setf slot-value-using-class) respectively. Other slot accessors use them internally. Thus, now we must define two methods for accessing and updating a slot of agent class. slot-value-using-class dispatches with an argument: slot-definition. Usual class uses pcl::standard-effective-slot-definition. We add cooperation-standard-effective-slot-definition as a subclass of pcl::standard-effective-slot-definition as following.\(^1\)

\(\text{(defclass cooperation-standard-effective-slot-definition}\)
\(\quad \text{#+CMUCL(pcl::standard-effective-slot-definition)} ; \text{if the system is CMUCL}\)
\(\quad \text{#-CMUCL(standard-effective-slot-definition)} ; \text{otherwise}\)

The following method is added for cooperation-standard-slot-definition. It is invoked during the class initialization process. In this case, a slot that has cooperative-strategy keyword parameter of a subclass of agent-class uses cooperation-standard-slot-definition as its descriptor.

\(\text{(defmethod pcl::effective-slot-definition-class}\}
\(\quad \text{((class agent-metalevel-class) initargs)}\}
\(\quad \text{(if (member :cooperation-strategy initargs)}\}
\(\quad \text{\quad (pcl::find-class ’cooperation-standard-effective-slot-definition)}}\}
\(\quad \text{\quad (pcl::find-class ’pcl::standard-effective-slot-definition)))}\}

The main difference is that the stored value in a slot is wrapped by the strategy object whose definition is described below. The first one is for value assignment.

\(\text{(defmethod (setf pcl::slot-value-using-class)}\}
\(\quad \text{(new-value t) (class agent-metalevel-class)}\}
\(\quad \text{\quad object (slotd comm-standard-effective-slot-definition)}\}
\(\quad \text{\;; assign the new-value to slotd of object in class}\}
\(\quad \text{\quad (let ((location (pcl::slot-definition-location slotd)))}}\}

\(^1\text{(setf slot-value-using-class) is a symbol whose name includes braces and a space.}\)
\(^\text{2In CLOS/MOP system, there is another slot definition class: standard-direct-slot-definition. Since it is parallel to standard-effective-slot-definition, we ignore it in this thesis.}\)
Separate Description of Communication Strategies

(slot-instance))
(cond ((typep (pcl::%instance-ref (pcl::std-instance-slots object) location) 'communication-strategy)
  (let* ((slot (pcl::%instance-ref (pcl::std-instance-slots object) location)))
    (update-value slot new-value) : model (history) is updated in update-value
    (when *out-of-local-computation-p*
      (funcall (set-strategy slot) object slot-instance slotd)))
  (t (setf slot-instance
       (make-instance (strategy-class slotd)
         :history (make-array (memory-size slotd) :initial-element 0.0)
         :value new-value))))

new-value)

The rest is for its reference.

(defmethod pcl::slot-value-using-class
  ((class agent-metalevel-class) object
   (slotd cooperation-effective-slot-definition))
  (let* ((location (pcl::slot-definition-location slotd))
         (strategy-object ...)) ;; same as the original method
    (if (typep (pcl::%instance-ref (pcl::std-instance-slots object) location) 'communication-strategy)
      (progn (funcall (ref-strategy strategy-object) slot)
        (value strategy-object)) ;; signal unbound error
      (pcl::slot-unbound class object (pcl::slot-definition-name slotd))))))

unboundness checking

In the initialization steps in make-instance, a strategy instance is created as the value of a slot. This means the standard method of slot-boundp returns t even if it has not been used. We define the following method for communication-strategy-slot-class. It call slot-boundp-using-class. Thus we add new method as well as slot-value-using-class, which invokes slot-boundp in the strategy object.
5.2.3 interface to programmer

Now we must provide an interface to problem-solving programmers. At first we provide a new macro for class definition: defagent. Its task is adding :metaclass option to class options and add some slots. Its definition is the following:

\[
\text{(defmacro defagent (class-name super-class-list &optional slot-defs &rest rest))}
\]

\[
\text{'(defclass .class-name}
\]

\[
\text{\quad .(if (member 'agent-class super-class-list))}
\]

\[
\text{\quad super-class-list}
\]

\[
\text{\quad (cons 'agent-class super-class-list))}
\]

\[
\text{\quad slot-defs}
\]

\[
\text{\quad ,0((cons '(:metaclass agent-meta-class) rest)))}
\]

Therefore the following form:

\[
\text{(defagent an-agent-class (}}
\]

\[
\text{\quad (cons (agent-class)}
\]

\[
\text{\quad \ldots)}
\]

\[
\text{\quad (metaclass agent-class) rest)))}
\]

\[
\text{\quad defagent can hide the names of both the metaclass and slots for communication strategies.}
\]

make-instance

In CLOS, making an instance of a particular class is performed by make-instance. Usually its arguments are for slot initialization.

Our communication strategies require knowledge about a communication cost. It depends on each environment. Thus when making an agent, we must give a cost function to the agent as well. However arguments required to the function depend on a strategy. Furthermore, an agent may use more than one strategies in our implementation. Thus we need a way to distinguish cost functions. Our solution is using a unique keyword for make-instance. The keyword is
Separate Description of Communication Strategies

built as a concatenation of 'cost-function' and the name of the strategy class. Thus if we use spatial-control as a strategy for exchanging the value in slot foo in user-defined class: C, and use temporal-control as a strategy for slot woo, its instance is created by:

(make-instance 'C 
   :communication-cost-for-spatial-control #'(lambda (num-of-packet) ...) 
   :communication-cost-for-temporal-control #'(lambda (frequency) ...) 
)

These keyword parameters are processed at initialize-instance method defined for agent-class. As we described earlier, the former lambda function is stored into cost-function slot of the instance of class spatial-control that is assigned to the slot: foo of the instance.

5.3 Applications

In this section, we investigate the applicability of the implementation by using two programs. The first one is the TSP program that we used in Chapter 4. The later is shared object management in distributed environments.

5.3.1 traveling salesman problem

With this framework, writing TSP program in a separated way is straightforward. Both the spatial connectivity control strategy and the frequency control strategy should be invoked when a slot renewal is occurred. The structure of the history depends on the strategy. Its definition is included in the definition of strategy classes.

(defagent search-agent ()
  ((threshold :slot definition 
               :set-strategy spatial-control) ...
  ))

(defclass spatial-control (communication-strategy)
  ((range :accessor range :initform 0 :initarg :range :type fixnum)
   (neighbor-list :accessor neighbor-list :initform nil)
   (recommended-size :accessor recommended-size :initform 1 :initarg :recommended-size :type fixnum))))
5.3. Applications

(defmethod set-strategy ((strategy spatial-control) instance)
  ;; range slot holds the new optimal range
  (setq (range strategy) (select-best-range (model strategy)))
  ;; send the body slot
  (send (make-instance 'spatial-control-message 
                  :body (list (class-name instance) 
                             (value strategy)))
        (choose-receivers (range strategy) (neighbor-list strategy))))

;; In the program that uses searching-agent,
(make-instance search-agent 
                 :communication-cost-for-spatial-strategy
                 #'cost-estimation-function)

(defmethod dispatcher ((class spatial-control) slot instance message)
  (if (spatial-control-class-message-p message)
      (when (betterp (body message) instance)
        ;; update-value is an internal method for communication-strategy
        (update-value class slot instance new-value)))

distinction of received data

In this program, a received threshold is assigned to the local threshold slot. If we used (setf threshold) for the assignment, a communication strategy would be invoked once more. It would cause repetition of communication. Though the loop, however, would be terminated since the utility of communication decreased by the increase of homogeneity of agents, if we want to avoid the loop, we should provide a raw assignment method. But the renewal should be a part of metalevel computation. Therefore receiving is not visible to the problem-level program but a task of metalevel computation invoked by the dispatching process at metalevel. It will use a raw level assignment method that does not invoke any communication strategy, which is called update-value in the above program.
5.3.2 shared object management

shared object management Most distributed programs use shared objects among processes. They require strong consistency. This is the most important difference from threshold in the previous example. From our point of view, threshold requires only weak consistency: the consistency is required for effective execution. Objects with strong consistency can be made in two ways in distributed environments. First approach is using virtual shared memory [Li86, Lh89]. The object is accessed by its address in the same way as local objects. In this system, memory pages are shared by all nodes. A copy of each page is located where it is accessed. Thus a running program has locality in memory access, it reduces the inter-node communication costs.

The second approach is based on a distributed algorithm. The object is implemented as local object on each node. They are identical: they have the same value. If a renewal is required, by broadcasting notification, all objects are updated as atomic action. This is a message-passing style implementation.

Some virtual shared memory approaches use parallel cache algorithm (i.e. Snoopy cache) to hold coherency among copies. This algorithm distribute copies at first step. The number of copies does not change. Thus it can be consider a variation of distribute algorithm based approach.

The main difference between these approaches is the number of identical copies. Most virtual shared memory approaches do not make copies of an object. Thus though the update’s cost is low, if the locality is low, the accessing costs will be high. On the other hand, the second approach requires broadcasting whenever the renewal occurs. We can imagine the adaptive approach: change the locations and the number of copies. These decisions should be based on the history of access/update pattern if we make it adaptive dynamically. Therefore our history based approach will be useful for this problem. Shared object management can be interpreted as a problem of cooperation. Note that it is rather MAS than DPS, since each agent’ local desires conflict with each other.

We should note that some papers investigate control method for changing the implementation of objects by program analysis (for example [KL95]). But it uses static, pre-execution
Here we describe a strategy that control the number of copies in a centralized manner [YNU95]. A pre-fixed node becomes the server. The others become owners or clients.

(defagent parallel-worker ()
  ((shared-object
    :cooperative-strategy shared-object-manager/central-manner)
   ...))

(defclass shared-object-manager/central-manner (communication-strategy)
  ((server-id :type host-id)
   (member-list :type list)
   (owner-p :type boolean)))

(defmethod server-p ((self shared-object-manager/central-manner))
  (= (server-id self) (id self)))

(defmethod set-strategy ((strategy shared-object-manager/central-manner) instance)
  (cond ((server-p strategy) (broadcast-update-request strategy)))

Figure 5.4: shared object distribution
(t (send-update-request strategy))

(defmethod ref-strategy [(strategy shared-object-manager/central-manner) instance]
  (cond ((owner-p instance) nil)
    (t (send-reference-request strategy)))

To receive messages, we need one more method. The following dispatcher should be
invoked from receive function.

(defmethod dispatcher [(strategy shared-object-manager/central-manner) instance message]
  (cond ((server-p strategy)
      (cond ((request-message-p message)
          (send-value strategy message))
        ((update-message-p message)
          (update-slot-internal strategy message)
          (broadcast-message strategy)
          (receive-all-ack strategy)))
        ((and (owner-p strategy) (update-request-p message))
          ;; update and return ack to server
          )
        (t (error)))))

To eliminate the central controller, each process should have a more cooperative protocol.
Modeling others that would use not only history of communication but also the result of
program analysis or the description of other agents' mental models will be required.

5.4 Discussion

As we explain above, this framework can describe many kinds of communication strategies.
Though we omit the detail here, we can also use this implementation to use temporary cache
method that is mentioned in Chapter 2. Now, we will investigate some demerits and difficulties
of this framework in the next section.

5.4.1 restriction of meta-object protocol approach

The main restriction of this implementation comes from MOP approach. MOP is not a reflective
computation mechanism that can change all of the base-level computation. Therefore an
agent can not change the role of itself by any communication strategy. If we want to let agent do it, more interface between a problem-level program and a metalevel program is required. But it spoils the merit of separate description. This restriction will be crucial when we want to built more autonomous agents that has a planner in it in order to decide its future plan.

5.4.2 role differentiation by method combination

On the other hand, some modifications to base programs can be done by method combination. For example, we have proposed the spatial connectivity control strategy on two structures. It requires an agent to change its role from a computing agent to a dedicated information collector. This changes is done as:

1. stop invoking original method.
2. register a new method for collecting information
3. inform this change to other agent

The main issue is how to change the method body. CLOS has a solution which use method combination. The change process is:

\[ \text{If I should become an information collector, I run a method for collecting information whenever I resume.} \]

\[ \text{If I should become a searching agent, I run a method for searching whenever I resume.} \]

This is a kind of method dispatching based on the result of the method priority computation. This method selecting computation differs from normal class-hierarchy based computation. But CLOS provides a way to define new, arbitrary method combinations.

In this framework, operators (method) that are described in an imperative style are invoked by the result of method selecting computation. By using a language extension, we can write it down in an imperative or declarative manner. When an agent A resumes to run a method \( M \), the method with the highest priority in all methods in \( M \) at A's point of view is selected, where the \( M \) that does nothing for collecting information is defined in a superclass of every agent class.
that is defined with `def-agent`. It is a default behavior of every agent. It is natural to inherit it and to select appropriate method dynamically by using method combination.

This framework will be implemented with `define-method-combination` and more protocol definition between application programmer and the system.

More drastic approach is mixing a MOPed OOPL with logic-based language extension. A planner-based description can be merged more flexibly. We plan the use of Scheme [CR91] with tiny-CLOS (subset of MOP in Scheme) [Kic91, Gal95] and an tool for logic programming. Since modern Scheme implementations support parallel programming, inter-processor communication, real-time GC and so on, they will be a good platform for DAI/MAS testbed.

### 5.4.3 efficiency

MOP based-languages has an issue; its efficiency. In languages that supports MOP, every slot access consists of a sequence of some methods. Comparing with a structure in traditional languages, an access to a field in the structure is transformed to a machine code for loading the content at an address with an offset. Sometimes the time for a reference of a slot becomes ten times slower than one for a reference of a field of a structure. But, as MOP researchers say, the language efficiency is not an issue of the language itself but one of implementation.

For example, CMUCL optimizes CLOS code if the metaclass used in a program is standard-meta-object. In our approach, since metaclass of slot is fixed during its execution, unfolding metalevel code into a sequence of flat procedure calls by compiler could be possible. In this case, we can avoid computation cost of MOP in execution time. For example, Masuhara et al. have proposed a method with partial evaluation of meta-level computation in ABCL/R2 and ABCL/R3 [MMWY92, MWIY92, MMAY95].

Anyway, flexibility to change the program's behavior is important in distributed systems, since programmers can not forecast its exact execution situation.

### 5.5 Summary

1. The communication strategies are implemented as a metalevel computation where the base program corresponds to the application program.
5.5. Summary

2. MOP is a useful method to extend a base language for this purpose.

3. Our communication strategies are thought as a set of the extensions to semantics of reference and assignment to an agent-local variable. Thus CLOS MOP provides a straightforward implementation of our strategies.

4. We illustrated the implementations of some communication strategies.

5. We showed some restrictions of our framework. One is caused by the restricted reflectional power of MOP that can not change the problem level programs. Another one is efficiency issue. We also showed some ideas to solve these issues.
Chapter 6

Conclusion

In this thesis, we presented communication strategies based on partial histories of agents for modeling their environment to select efficient communication structures dynamically. The experiments showed that the strategies brought good communication structures to autonomous agents. We proved that agent's local history as a qualitative model of revision of information, which is a measure of execution, is useful for MAS.

Though there are some restrictions such as the implemented strategies assuming homogeneity of agents, the simplicity of combining some pieces of information, and the information of the communication cost function, we think that these strategies can be used in many applications that acquire new information in execution time, if we can find information which value increases monotonously and can build a communication cost function. These would contain distributed database systems that replicate data.

The reason of good results from local-history based estimation method is that agents in systems we used can be assumed homogeneous. Thus we need not to build an exact model of other agents. This implies the assumption that their goals do not conflict with each other. It is not useful for MAS architecture but DAI. The history-based expectation mechanism, however, could apply to not only homogeneous environments but also heterogeneous ones. The history of the revision of information at other agents in current implementations is not separated from its own history. Agents exchange the information with each other and update its history with the received information. But in heterogeneous network, the operation of histories would become
an important problem. The implementation and the evaluation of strategies based on separated histories are a future plan. It will also work on heterogeneous agents.

In this thesis, we proposed a cooperation scheme on distributed systems. However cooperative processing is not only useful on distributed systems but also on concurrent systems. One example that requires cooperation, even if the communication cost can be ignored, is a parallel search based on genetic algorithms. A search process would fall in a local minimum position by over-distribution of the best code at each step. Thus broadcasting the best code pattern does not necessarily lead to the optimal form of computation. The same discussion is held on heuristic-base parallel search. In these examples, cooperation changes the way of sharing information between agents. Thus we can define cooperation as methods for selecting an appropriate structure of processing elements. Its application is not restricted to distributed systems. We think that local-history based methods like ours will be useful for building schemes on such systems.

Our study assumes that the intention of a programmer is presented in a program but not a specification. Therefore we consider about neither rule-based or operator-based description nor MAS. Thus our approach has a restriction necessarily for building strongly autonomous systems. But we think there is a hierarchy of autonomy. At the top of the hierarchy, the unit is human or a very autonomous agent. They can be described well by the term of belief, desire, and intention (they are called as BDI theory). But the bottom level the description of the goal of a computational unit is decomposed to a sequence of orders. If we assume the number of computation unit in the real world becomes very large, to use them effectively, pulling parallelism off is the important issue, even if the platform is distributed and thus we can not ignore the communication cost on them. Since there will be a hierarchy of physical closeness of computational units, our approach would be used in the low-level of problem-solving strategy’s hierarchy.

Of course, history-based estimation method will be useful in high-level planning. It will require probabilistic reasoning. The emergence of intelligence is one of the important research themes in Artificial intelligence. Recent studies about emergent computation investigate a way to make information processing machine from a pool of simple units [For91, FM90]. Making
more intelligent system from units by a kind of evolution requires a meta-calculation about utility like stability or uniqueness. Therefore selecting the input of itself and selecting related modules is crucial. The issue about computation of connectivity between the units will be emerged once again [Lan90, KHH89]. Therefore more study about connectivity should be expected.
Acknowledgments

I would like to thank to Professor Kazuo Ushijima at Kyushu University for supporting this research and improving the quality of this thesis. I am very grateful to Professor Akifumi Makinouchi, Professor Ryuzo Hasegawa and Professor Kotaro Hirasawa at Kyushu University. Their valuable comments are gratefully acknowledged.

I would like to express my appreciation to Associate Professor Norihiko Yoshida at Kyushu University for discussion about DAI, MAS, OOPL, and direction of this research for long years.

I would also like to note two graduate students: Hiroomi Yamamura, the implementer of the frequency control strategy and Kenji Yamasaki, who implemented the core part of CLOS MOP-based system.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


Index

A* search, 68
agent, 2
agent-meta-class, 75
bounded rationality, 2
class, 74
communication-strategy, 77
defagent, 81
Distributed Artificial Intelligence, 2
Distributed Hearsay-II, 1
Distributed Problem Solving, 3
Distributed Vehicle Monitoring Testbed, 10
ergodic process, 26
instance, 74
metaclass, 74
MetaObject Protocol, 74
Multi-Agent System, 3
NK-model, 23
objects, 74
Pursuit Problem, 13
shared object management, 68, 84
Traveling Salesman Problem, 53
utility of communication, 22
virtual shared memory, 84
業績一覧

論文

1. 吉田紀彦、槇崎修二、下川俊彦，“協調処理モデルCellulaの分散処理系と支援環境”，並列処理シンポジウムJoint Symposium on Parallel Processing '90論文集（平成２年 5月）

2. 吉田紀彦、槇崎修二，“場と一体化したプロセスの概念に基づく協調処理モデルCellula”，情報処理学会論文誌第31巻第7号（平成２年 7月）


4. 吉田紀彦、槇崎修二，“協調処理モデルCellulaの分散処理系”，情報処理学会論文誌第32巻第7号（平成３年 7月）

5. 槇崎修二，“分散問題解決のための自律的構造選択戦略”，日本ソフトウェア科学会MACC93論文集「マルチエージェントと協調計算 III」（平成４年 10月）


7. 槇崎修二，“追跡問題における視界の影響”，日本ソフトウェア科学会MACC94論文集「マルチエージェントと協調計算 IV」（平成７年 11月）

講演

1. 植崎修二・古賀慎一郎・谷口倫一郎・河口英二，“知的機能支援システム IPSSENS-II におけるテクノ管理方式について”，情報処理学会九州支部大会（昭和 63 年 3 月）

2. 吉田紀彦・植崎修二，“場と一体化したプロセスの概念に基づく並列協調処理モデル”，電子情報通信学会技術研究報告 CPSY 89-19（平成元年 8 月）

3. 植崎修二・吉田紀彦，“並列協調処理モデル Cellula の実現と評価”，日本ソフトウェア科学会第6 回大会（平成元年 10 月）


5. 植崎修二・堀田英一，“通信サービス記述のための知識と動作に基づく様相論理 SSL”，第 40 回情報処理学会全国大会（平成 4 年 3 月）

6. 植崎修二・堀田英一，“通信サービス記述のための知識と動作に基づく様相論理 SSL”，信学技報 COMP（平成 4 年 5 月）

7. 植崎修二・堀田英一，“通信サービス記述のための知識と動作に基づく様相不動点論理 SSL”，情報処理学会研究会報告 PRG（平成 4 年 8 月）

8. 植崎修二・吉田紀彦，“大規模分散計算環境下における分散探索エージェントの通信戦略”，日本ソフトウェア科学会第 10 回大会（平成 5 年 6 月）

9. 植崎修二・吉田紀彦，“大規模分散計算環境における分散探索エージェントの通信戦略”，情報処理学会研究会報告 AI-93-89（平成 5 年 8 月）

10. 植崎修二・吉田紀彦，“大規模分散計算環境における分散探索エージェントの通信戦略”，支部省重点領域研究「超並列原理に基づく情報処理基本体系」第 3 回シンポジウム（平成 5 年 9 月）
11. 樋崎修二、吉田紀彦、"自己組織化エージェントによる協調処理"、文部省重点領域研究「超並列原理に基づく情報処理基盤研究」第4回シンポジウム（平成6年3月）

12. 田村広臣、樋崎修二、平島和夫、"協調探索におけるエージェント間の通信頻度決定戦略"、電気関係学会九州支部連合大会（平成6年9月）

13. 山崎賢治、樋崎修二、平島和夫、"メタオブジェクトを用いた分散問題解決プログラムの分離記述"、1995年電子情報通信学会全国大会（平成7年3月）

14. 山崎賢治、樋崎修二、平島和夫、"メタレベル計算を用いた協調処理の実現"、情報処理学会研究報告PRG95-82（平成7年8月）