Paradigms for Parallel Distributed Programming [1]

F. Araque, M. Capel, A. Palma
Dpt. Lenguajes y Sistemas Informáticos
E.T.S. Ingeniería Informática
Universidad de Granada
Avda. Andalucía 38
18071 Granada - Spain
{faraque, mcapel, apalma}@goliat.ugr.es

J.M. Mantas
Dpt. Informática
E.P. Superior
Universidad de Jaén
Avda. Madrid 35
23071 Jaén - Spain
jmantas@apolo.ujaen.es

Abstract

The work proposes a new classification of parallel algorithm schemes to program multicomputer systems, these schemes being called paradigms. The proposed classification is intended to improve the programmability and portability of distributed parallel algorithms derived with these paradigms. In order to design algorithms and to implement applications based on these paradigms, a language that directly supports them has been implemented on the Occam- Transputer platform. This language is based on a class of distributed active objects (*ODA*s) allowing a flexible and safe implementation of different algorithms.

1 Introduction

One of the most importat utilities of multicomputers is their use in algorithmic computation to obtain speed-up and efficiency. These architectures present advantages due to their quasi-unlimited scalability and low cost, if compared to multiprocessors. Nevertheless, it is well known that the difficulties of producing the software for these architectures is delaying their widespread use with current applications. We propose a methodology for Distributed Programming based on applying an *informal* derivation process from a set of *programming paradigms* to obtain an *algorithmic skeleton* and finally a distributed algorithm which solves a particular problem. We intend the proposed *paradigms* to be as general as possible and supported by the wider range of parallel programming languages. Nevertheless, our research work was motivated by our real necessity of overcoming the problems derived from our experience in programming with the Occam language, therefore the work presented is influenced by this language.

Several classifications of parallel algorithm paradigms have been proposed in the last few years. Among the most significant, with respect to our work, are those proposed by G.A. Andrews [1], P.Brinch-Hansen [2] and F.Rabhi [3]. Andrews carried out a clarifying study of several distributed programming techniques applicable to numerous practical problems. These techniques assume a different communication pattern among the processes in order to obtain a specific distributed algorithm, therefore each one is a *model* for programming distributed

[1] This work has been financed by the project TIC94-0930-C02-02 of the Comisión Interministerial de Ciencia y Tecnología

algorithms. Andrews termed these models: *paradigms for process interaction in distributed programming.*

In spite of the quality of Andrews' work, several practical drawbacks appear when using the aforementioned *paradigms*. The most important are:

- it is difficult to guess what is the *algorithmic skeleton* common to all the algorithms of the same paradigm
- the proposed paradigms are too problem dependent and do not give practical insight to deriving new algorithms
- these *paradigms* are better understood as *model programs* than as authentic paradigms.

Therefore, in addition to the above paradigms, it is necessary to define an intermediate design level between the vague description of a *programming paradigm* and a set of *model programs*. This intermediate level is a *generic program* or *algorithmic skeleton* which describes the common control structure shared by all the algorithms belonging to the paradigm. Neither the data types nor the code of specific data-dependent procedures need to be detailed at this level, since these depend on concrete programs.

The P.Brinch-Hansen [2] proposal is more in accordance to the proper concept of *programming paradigm* in the sense established by R.W.Floyd [4] several years ago. The concepts of *paradigm, algorithmic skeleton, model program*, etc. are defined precisely; to understand the role these concepts play in a systematic derivation of distributed algorithms and programs, numerous examples were presented. Almost all of the aforementioned concepts will be assumed in the remainder of this work, only a subtle difference in our concept of *parallel paradigms* makes the theoretical background of the two proposals different. In P.Brinch-Hansen's own words, the golden rule to understanding the derivation process of distributed algorithms using *parallel paradigms* is: *the essence of the (distributed) programming methodology is that a model program has a component that implements a paradigm and a sequential component for a specific application.*

Our scheme considers a *parallel paradigm* as a class of algorithms which have a similar communication structure intimately related with the global data dependencies among the processes in which are these algorithms decomposed. It is of minor importance whether the distributed algorithms belonging to the same paradigm class have a similar control structure in their corresponding sequential algorithms or not.

A SPMD programming notation is assumed in our scheme. The processes in a distributed program have a symmetric code which probably operates in different partitions of global shared data types. The difference with the pure SPMD programming style is that in our case the code of the processes may be specialised depending on initial parameter values instantiated at the program *configuration phase*.

The main objective of our research is to obtain efficiency in distributed applications by deriving *quasi-optimal* communication structures (named *logical topologies*) which minimize global data dependencies among the distributed processes and grants, at the same time, better *load balancing* among the processors of a multicomputer. It is well known that the distributed applications efficiency depends in a negative manner on the number of accesses to global shared data. Our scheme tries to solve this problem by allowing the programmer to flexibly define logical topologies according to a particular application, where the remote data accesses are minimized as much as possible. To do so, we require that the *sequential component* of the distributed algorithms have a minimum of dependencies with the communication process code, therefore allowing an *algorithmic skeleton* to be adapted to different concrete applications whose algorithms could belong to the same paradigm.

In addition to the above-stated objective we also intend to apply the *paradigm-based* design method to improve the *reusability* of the programs derived for multicomputers [5].

Finally, it is necessary to stress the fact thet we do not define recursive parallel constructions nor dynamic creation of objects (processes, channels) in our programming notation, therefore the compatibility with the Occam process model is maintained. The reason is that the majority of programming languages for multicomputers today do not allow the mixing recursion and parallelism and therefore we do not think it appropiate to have parallel programming paradigms which are not completely supported by our programming languages.

The paper is organised as follows: first we propose a methodology for programming distributed algorithms, based on applying different *algorithmic skeletons* which are described in one of the proposed *parallel paradigms* in order to derive *model programs* in the *ODA* based language. Second, we present an implementation language, based on *ODAs*, which fully *supports* the parallel paradigms presented. In this language a complex application to solve the TSP with a distributed parallel algorithm is presented. Finally, the conclusions of the present work are dicussed.

2 Classification

In order to clarify the concepts, the *algorithmic skeletons* of the proposed paradigms are presented below. An Occam2-style pseudocode syntax is selected to present the *algorithmic skeletons*. The parallel processes will first be specified by Occam procedures, followed by the configuration code which specifies the parallel components of the application and the connections between these components.

According to the aforementioned presentation scheme, every Occam procedure represents a generic process of the program and includes as formal parameters the channels needed to communicate with other processes. These communications may be carried out with processes placed on the same processor (local communications) or placed on remote processors (remote communications). The *remote* and *local* channels are not distinguished, all are declared as *formal channel parameters* in the procedure's head, but the local or remote communication *character* of each channel may be inferred by process code observation. The channels needed to connect the different components (according to the communication structure of the ODAs and to the use relationships) are declared and specified as actual parameters for the procedures which implement the different components when each *skeleton* is instantiated to solve a specific problem. These details must be taken into account in a later design stage of a particular program and therefore they are not addressed now.

2.1 Master-slave

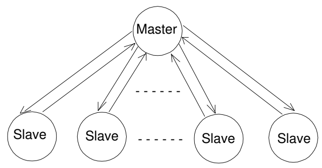
The *master-slave* paradigm consists of a number of independent tasks performed by a set of *slave* processes under the centralized control of a *master* process. These tasks may be either of the same type (every slave process executes the same code with different data) or of different types (the slaves processes' code can be different). There is no data dependency between the different tasks performed by the slave processes; therefore, these tasks can be executed asynchronously, and there is no need for direct communication between the slave processes. The slave processes interact with the master process by receiving work units from the master and by sending the results of the computation performed to the master.

The master process holds centralized control of the algorithm, which includes the delivery of tasks between the different slave processes, workload balancing in the distributed algorithm, collecting results from the slave processes, and implementation of global control operations, such as termination detection, etc.

The parallelizing of algorithms according to this paradigm is carried out by replicating the purely algorithmic sequential code between the different slave processes. The replication of the algorithmic code over the slave processes implies the need for distributing data and implementing global control operations. The code of the master process must implement these issues in a centralized way, as described above.

The communication structure for this paradigm must provide a connection between the master process and every slave process.

An algorithmic skeleton for distributed algorithms with the master-slave paradigm is shown. The computation begins with the delivery of initial work for the slave processes. The local computations in the slave processes are carried out in parallel with the requesting of new units of work in order to avoid having the slave processes idle while receiving work from the master process.




```
master.process ([]chan of m.s.protoc to.slaves, from.slaves)
  generate initial work for slave processes
  parallel any.slave = 0 for number.of.slaves
    to.slaves[any.slave]  !  initial.work.to.slave.process
  while active
    alternatives any.slave = 0 for number.of.slaves
      from.slaves[any.slave] ? request.from.slave.process
        to.slaves[any.slave] ! work.to.slave.process
  parallel any.slave = 0 for number.of.slaves
    from.slaves[any.slave]  ?  final.results.from.slave.processes
:
slave.process (chan of m.s.protoc  from.master, to.master)
  from.master  ?  initial.work.from.master.process
  while active
    parallel
      perform.work ()
      to.master   !  request.to.master.process
      from.master ?  work.from.master.process
  to.master  !  final.results.to.master.process
:
-- configuration: execute master and slave processes in parallel
[number.of.slaves]chan of m.s.protoc master.to.slave, slave.to.master:
parallel
  master.process (master.to.slave.ch, slave.to.master)
  parallel i=0 for number.of.slaves
    slave.process (master.to.slave[i], slave.to.master[i])
```

2.2 Peer to Peer

There is no data dependence between the processes in this paradigm. The processes asynchronously attempt to solve either different complete problems or independent parts of the same problem. The interaction between the different processes is minimal since to each process can execute independently of the other processes. Sometimes it may be interesting for the processes to comunicate each other in order to exchange useful data for solution of the problem. This kind of interaction is very specific for the algorithm being parallelized, but in any event the processes interact peer to peer, without any centralized control. The parallel algorithm may terminate either when any of the parallel processes terminates or when all of them terminate, and the final solution may be obtained from the different solutions achieved by the peer processes.

As an example, the following algorithmic skeleton could be applied to a set of processes which execute a sequential algorithm code and, at the end of each iteration, exchange with their neighbors in a ring topology a local value to optimize their local solutions. In order to allow independent execution of the processes, the exchange of this value is done asyncronously. Asynchronous message passing would be the most suitable communication mechanism in this case, although it can be simulated using synchronous message passing by means of a buffer process.

```
peer.proc (chan of p2p.protoc from.left, to.left, from.right, to.right)
  chan of p2p.protoc output.to.buffer, input.from.buffer:
  parallel
    asyncronous.buffer (output.to.buffer, input.from.buffer,
                        from.left, to.left, from.right, to.right)
    while active
       perform.work ()  -- algoritmic sequential code of the process
       -- exchanging of values with neigbours in the ring:
       output.to.buffer  !  local.exhange.value
       input.from.buffer ?  (left.exchange.value, right.exchange.value)
:
-- configuration: execute peer processes in parallel
[number.of.peers]chan of p2p.protoc clockwise, anticlockwise:
parallel i = 0 for number.of.peers
  peer.proc (clockwise[i], anticlockwise[i], anticlockwise[i+1], clockwise[i+1])
```

2.3 Client-server

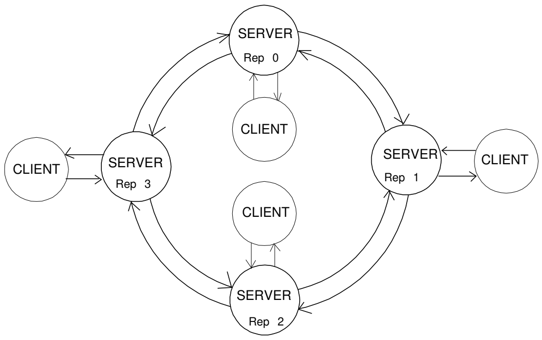
In the *client-server* paradigm there are two classes of processes: *clients*, which perform requests which trigger reactions on server processes, and *servers*, which accept requests for the performance of a service not available in client processes.

In this scheme, particularized to the SPMD framework, some processes may act as client and server at different moments according to the interactions performed with other processes. This scheme is similar to the replicated servers scheme proposed in [1]. The global data and resources are distributed by partition, replication, etc., the implementation of mechanisms of workload balancing being necessary. The control is fully distributed: there is no central process; this may imply the implementation of distributed protocols of synchronization operations between the replicated servers, distributed mutual exclusion, distributed data consistency, termination detection.

The parallelizing of algorithms according to this paradigm is carried out as follows:

- by replicating the code of the processes executing the purely sequential algorithmic code which depends on the algorithmic technique being parallelized,
- the distributed implementation of the global data and control operations is carried out by a set of processes which act as servers for the application processes: when an application process needs any operation on these global data, it makes a service request to the server process directly connected with it,
- when a service is not locally available in a replicated server, the server process re-sends the service request to another server process that can service the request, acting in this case as a client,
- the topoloy of the communication structure must allow communication between the different application processes and their respective servers, and between different replicated servers.

An algorithmic skeleton for a distributed algorithm based on the client-server paradigm follows. The replicated servers are connected by a ring topology.




```
application.process (chan of c.s.protoc  request.ch, reply.ch)
  while active
    ...
    request.ch ! service.request.to.server.process
    reply.ch ? reply.from.server.process
    ...
:
replicated.server (chan of c.s.protoc  request.from.appl.proc, reply.to.appl.proc,
           chan of c.s.protoc  request.from.other.servers, reply.to.other.servers,
                               request.to.other.servers, reply.from.other.servers)
  while active
    alternatives
      request.from.appl.proc  ?  service.request.from.application.process
        if  service.locally.available
          then  results := perform.service ()
                reply.to.appl.proc  !  results
          else  manage fail in providing service as appropiate
      request.from.other.servers  ?  service.request.from.other.server.process
        -- the same code as previous alternative,
        -- reply is sent back through 'reply.to.other.servers' channel
      service.request.to.other.servers.needed  &  skip
        request.to.other.servers  !  service.request.to.other.server.process
      reply.from.other.servers ? reply
        if  reply = results   -- a requested service was available in other server
          then  update local state of replicated.server
          else  manage fail in requesting service as appropiate
      reply.available.for.waiting.client.process  &  skip
        if client = application.process
           then  reply.to.appl.proc  !  reply
           else  reply.to.other.servers  !  reply
:
-- configuration:
[number.of.replicas]chan of c.s.protoc appl.proc.to.server, server.to.appl.proc:
[number.of.replicas]chan of c.s.protoc cli.ser.request, cli.ser.reply:
parallel
  parallel i = 0 for number.of.replicas
    replicated.server (appl.proc.to.server[i], server.to.appl.proc[i],
                       cli.ser.request[i], cli.ser.reply[i],
                       cli.ser.request[i+1], cli.ser.reply[i+1])
  parallel j = 0 for number.of.replicas
    application.process (appl.proc.to.server[i], server.to.appl.proc[i])
```

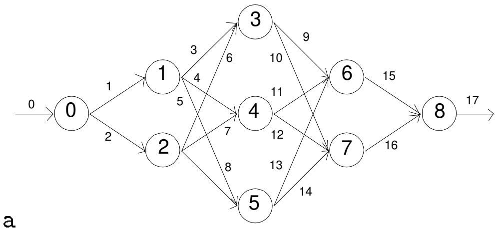
2.4 Data dependent

In this paradigm, the existing processes show the relationships between the different tasks performed during the execution of the algorithm. The results of some tasks are the input data of others. A stream of data flows through the diferent processes which transform it, branch it, merge it, sort it, etc.

The relationships between the processes appear as a result of the implicit data dependencies in the algorithm. For example, in a sort-merge algorithm performed by a tree of processes, the internal nodes of the tree are waiting for sorted pieces of data from their sons, and these pieces are merged into a single piece which is sent to the father in the tree.

The processes in the distributed algorithm make up a network of processes whose topology shows the data dependence relationships between the different processes. Examples of these topologies include pipelines, trees, cubes, meshes, rings, etc.

As an example, consider a network of filter processes consisting of a number of stages, such that data flow through the different stages. Every process in stage i iteratively receives data from any process in the previous stage $i-1$, transforms these data, and sends them to a process in the next stage $i+1$. The topology required for this process network connects every process with all the processes in its respective previous and following stages.




```
filter.process ([]chan of d.d.protoc input.ch,
                []chan of d.d.protoc output.ch)
  while active
    alternatives i = 0 for num.of.input.channels
      input.ch[i]  ?  input.data
        transform input.data into.output.data
        chose channel index j to send output.data
        output.ch[j] ! output.data
:
-- configuration
[18]chan of d.d.protoc data.ch:
parallel
  filter.process ([data.ch[0]], [data.ch[0], data.ch[1]]) -- stage 0
  filter.process ([data.ch[1]], [data.ch[3], data.ch[4], data.ch[5]]) -- stage 1
  filter.process ([data.ch[2]], [data.ch[6], data.ch[7], data.ch[8]]) -- stage 1
  filter.process ([data.ch[3], data.ch[6]], [data.ch[9],  data.ch[10]]) -- stage 2
  filter.process ([data.ch[4], data.ch[7]], [data.ch[11], data.ch[12]]) -- stage 2
  filter.process ([data.ch[5], data.ch[8]], [data.ch[13], data.ch[14]]) -- stage 2
  filter.process ([data.ch[9], data.ch[11], data.ch[13]], [data.ch[15]]) -- stage 3
  filter.process ([data.ch[10],data.ch[12], data.ch[14]], [data.ch[15]]) -- stage 3
  filter.process ([data.ch[15], data.ch[16]], [data.ch[17]]) -- stage 4
```

3 A Language for the Implementation of Parallel Algorithms

Our experience in programming with Occam-2 has shown us that this language is not adequate to carry out a systematic translation of the *algorithmic skeletons* presented in the above section to concrete programs, mainly due to the following disadvantages:

- The Occam channels are a very *low level* support to implement an interface for program modules which encapsulate operations and other *services* needed by other processes or *components* of the program.

- The despcription of the communication structures of the program is spread over different places and stages of the application building.

- The user must write a configuration description file according to the communication structures of the processes and to the use relationships between the components of the application. Therefore, he is obliged to program the aforementioned communication structure in a different context from the main program text of his application.

The above-mentioned drawbacks and other problems have led us to consider a different programming language more adequate for deriving distributed programs in which data abstraction and reusability facilities allow easier translation from paradigms to distributed programs in order to solve specific problems.

3.1 Distributed Active Objects: Encapsulation of paradigm-dependent code in a structured and reusable way

The parallelizing of algorithms under the different paradigms seen above is carried out more easily if we differentiate two parts in the design of a parallel algorithm:

- From the sequential version of the algorithm, a replicated application process is obtained, whose code maintains the main structure of the algorithm.

- The implementation of global data and communication requirements can be easily derived from the algorithmic skeleton corresponding to the specific paradigm being applied.

Instead of implementing the needed communications directly in the application processes code and declaring the channels which would connect them directly, the sequential application processes are maintained as generic as possible in order to promote reusability. The global data and the communications of the distributed algorithm are encapsulated in a class of modules called Distributed Active Objects (ODAs). An ODA module is an object with independent execution from the application processes, whose code is also replicated over the nodes of the multicomputer. The application processes interact with the ODAs by calling to their operations. The operations of an ODA may involve communication between the distributed parts of the algorithm; these remote communications are encapsulated and implemented by communications between the replicas of the ODA. The way in which the replicas of each ODA connect and communicate with one another comprise the object's communication structure, which depends on the applied paradigm of parallel programming.

3.2 ODA description language

A language to describe ODA modules has been proposed in order to overcome Occam limitations in providing the necessary encapsulation and abstraction mechanisms. In the notation of the language, the description of an ODA module is carried out in two separate parts:

Definition Part:

```
ODA.DEFINITION name ({0, header.parameter})
  [CONST
    {abbreviation}]              <--- as in Occam
  [PROTOCOL
    {protocol.definition}]       <--- as in Occam
  DIFUSION.MESSAGES
    {channel.declaration}        <--- as in Occam
  TOPOLOGY
    {connection.statement}
  SHARED.OBJECTS
    {call.declaration}
```

The parameters in the header and the abbreviations defined in the `CONST` section may be used to specify values which are fixed at configuration time. The communication structure of the ODA is described, by declaring, locally to each replica (`DIFFUSION.MESSAGES` section), a set of channels which allow communication with other replicas by specifying (in the `TOPOLOGY` section) connections of these channels according to the topology of the communication structure:

```
connection.statement  ::=   CONNECT channel TO replica.number
```

The interface for the ODA operations is specified, using a similar notation to the remote call channels of Occam 3 [6]:

```
call.declaration     ::=   CALL name ({, formal}) :
formal               ::=   VAL type.specifier {1, name}
                       |   RESULT type.specifier {1, name}
```

Implementation Part:

```
ODA.IMPLEMENTATION name ()
  USE ODA name :
  SHARED.OBJECTS
    [INIT
      {sentence}]
    PORT
      {alternative}
```

```
alternative      ::=  guarded.sentence
                        sentence
guarded.sentence ::=  boolean, SKIP
                 |  [boolean, ] receive.sentence
                 |  [boolean, ] accept.sentence
accept.sentence  ::=  ACCEPT name ({, formal}
                        sentence
```

The ODA-replica code that implements the operations for the ODA is described by means of a `PORT` construction, which has the semantics of the iteration of a non-deterministic selection over a set of alternatives.

The alternatives of the `PORT` construction implement the remote communications between the different replicas by means of `SEND` and `RECEIVE` commands over the channels declared in the `DIFFUSION.MESSAGES` section. The interaction with users is described by means of `ACCEPT` commands over the calls declared in the `SHARED.OBJECTS` section of the definition part of the ODA.

The application processes code is described in a separate module, in which each ODA to be used is *declared* (the interface described in its definition part is imported in order to be able to call to its operations), and *instantiated* (actual values are assigned to the header parameters of the ODA definition, and an implementation part compatible with the definition, that is, assuming the same interface, is specified).

3.3 Application: An ODA-based program for a parallel Branch and Bound algorithm

A Branch and Bound (BB) algorithm which finds an optimal solution to the Traveling Salesman Problem (TSP) has been parallelized according to the proposed paradigms. The TSP could be represented by a directed graph consisting of a set of vertices (cities) and labeled edges (distances between cities). One solution for the TSP is a path in which all the cities are visited exactly once. An optimal solution is a solution with minimal cost.

The BB algorithm dynamically builds a search tree, whose root is the initial problem and its leave nodes are solutions to the problem. For a node in the seach tree, two independent subproblems are generated by choosing an edge $< i, j >$. The left son is built by including the edge $< i, j >$ in the path of the solution, and excluding the other remaining edges $< i, ? >$ and $<?, j >$. The right son is built by excluding the edge $< i, j >$ from the path of the solution.

The way in which a parallel implementation of the BB algorithm is achieved is closely related to the search tree construction. A solution of the TSP problem is achieved by successive application of the division procedure described above. Consider a network of proceses divided in stages. Processes in stage i are specialized in dividing subproblems of size $N - i$, the size of a subproblem being the number of cities not yet included in the path of a solution and N the number of cities in the initial problem. Therefore, the processes in stage i divide the subproblems received from the processes in stage $i-1$, and send subproblems of size $N-i-1$ to the processes in the next stage. The processes in the last stage of the network divide subproblems of size 2 and find solutions for the TSP.

The algorithm can be implemented in the described manner by a pipeline with $N-1$ proceses, but this results in bad work-load balancing, since processes in intermediate stages would be overloaded. In order to adapt the topology of the process network to the actual structure of the search tree, processes in the intermediate stages are replicated in order to achieve better load-balancing. The number of processes in each stage is set according to the density of subproblems of the corresponding size in the seach tree. The topology of the network is defined in a generic manner by declaring the number of replicas per stage as a parameter whose actual value is specified at configuration time.

```
ODA.DEFINITION data.dep (VAL []INT replicas.per.stage)
  #USE "ddfunctions.lib"
  CONST
    number.of.stages            =  SIZE (replicas.per.stage) :
    stage                       =  position (REP.ID, replicas.per.stage) :
    first.replica.of.prev.stage =  first.replica (stage-1, replicas.per.stage) :
    first.replica.of.next.stage =  first.replica (stage+1, replicas.per.stage) :
  PROTOCOL
    BB.protocol IS ... :
  DIFFUSION.MESSAGES
    [replicas.in.prev.stage]CHAN OF BB.protocol from.prev.stage:
    [replicas.in.next.stage]CHAN OF BB.protocol to.next.stage:
  TOPOLOGY
    IF current.stage <> 0
      DO i = 0 FOR replicas.per.stage[stage-1]
        CONNECT from.prev.stage[i] TO first.replica.of.prev.stage + i
    IF current.stage <> (number.of.stages-1)
      DO i = 0 FOR replicas.per.stage[stage+1]
        CONNECT to.next.stage[i] TO first.replica.of.next.stage + i
  SHARED.OBJECTS
    CALL Get.Node (RESULT ... node ) :
    CALL Add.Node (VAL ... node) :
    CALL Prune (VAL INT upper.bound) :

ODA.IMPLEMENTATION data.dep.implementation ()
  USE ODA data.dep:
  SHARED.OBJECTS
    PORT
      REP i=0 FOR replicas.in.previous.stage
        RECEIVE (from.prev.stage[i], node)
          store node in local list
      local.list.not.empty, ACCEPT Get.Node (RESULT ... node)
        node := take node from local list according to search heuristics
        SKIP
      ACCEPT Put.Node (VAL ... node)
        SKIP                                                  -- empty body
        IF size.subproblem(node) = (N-stage)
          THEN  store node in local list
          ELSE  choose channel index i for sending node to next stage
                SEND (to.next.stage[i], node)
      ACCEPT Prune (VAL INT upper.bound)
        store upper.bound
        remove nodes from local list with lower.bound > upper.bound
```

When a sub-optimal solution is found its cost is considered as an upper bound to the cost of the final solution of the TSP. New upper bound values should be communicated to all the processes in order to remove the subproblems with lower bound values greater than the upper bound. The communication of global upper bound values should be made asynchronously and consistency of the different upper bound values in the processes is not required. The implementation of the distribution of global upper bound values is carried out by an ODA according to the peer to peer paradigm.

```
ODA.DEFINITION p2p.ring ()
  DIFFUSION.MESSAGES                 -- connect different replicas
    CHAN OF INT receive.chan, send.chan:
  TOPOLOGY
    CONNECT receive.chan, send.chan TO (REP.ID-1)\NODES, (REP.ID+1)\NODES:
  SHARED.OBJECTS    -- operations available to application processes
    CALL read (RESULT INT value) :
    CALL write (VAL INT value) :

ODA.IMPLEMENTATION p2p.ring.implementation ()
  USE ODA p2p.ring:
  SHARED.OBJECTS
    PORT
      RECEIVE (receive.chan, new.value)
        IF new.value < local.value
          THEN local.value := new.value
               SEND (send.chan, local.value)
      ACCEPT read (RESULT INT value)
        value := local.value                        -- accept statement body
        SKIP
      ACCEPT write (VAL INT value)
        new.value := value                          -- accept statetemt body
        IF new.value < local.value
          THEN local.value := new.value
               SEND (send.chan, local.value)
```

The `APPLICATION` code contains the branch and bound sequential code and the declaration and instantiation of the ODAs.

```
APPLICATION parallel.branch.and.bound
  NODES is ...:              -- number of replicas
  VAL replicas.per.level IS [1,4,8,16,32,64,...,1]:
  ODA data.dep work.oda:
  ODA p2p.ring upper.bound.oda:
  work.oda (replicas.per.level) IMPLEMENTED data.dep.implementation ()
  upper.bound.oda () IMPLEMENTED p2p.ring.implementation ()
  SEQ         -- application process code
    WHILE active
      {work.oda}Get.Node (node)
      select edge <i,j> to split node
      left.son.node := include.in.path (<i,j>, node)
      IF is.a.solution(left.son.node)
        THEN upper.bound := min (upper.bound, cost(left.son.node))
             {upper.bound.oda}write (upper.bound)
        ELSE {work.oda}Put.Node (left.son.node)
      right.son.node := exclude.from.path (<i,j>, node)
      IF is.a.solution(right.son.node)
        THEN upper.bound := min (upper.bound, cost(right.son.node))
             {upper.bound.oda}write (upper.bound)
        ELSE {work.oda}Put.Node (right.son.node)
      {upper.bound.oda}read (upper.bound)
      {work.oda}Prune (upper.bound)
```

4 Conclusions

The difficulties of producing the software for multicomputers is delaying their generalized use as the standard platform for current distributed and cooperative applications. One of the most important research goals, in order to make advances in the development of multicomputers software, is therefore the development of methodologies and tools which will allow algorithms and distributed programs to be derived systematically and reliably. In order to do so, the following solutions have been proposed in this paper:

1. A method for deriving distributed algorithms from a set of newly proposed programming paradigms.
2. A methodology for programming distributed algorithms based on hiding the global shared data and the remote communications in distributed applications from the user's process code. The sequential component of the distributed algorithms derived turns out to have a minimum of dependencies with the communication related code.
3. A new language based on a class of distributed active objects termed ODAs which flexibly support the paradigms referred to in (1).

References

[1] G.R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 15(1): 49-90, 1991.

[2] P. Brinch-Hansen. Model Programs for Computacional Science: A Programming Methodology for Multicomputers. *Concurrency: Practice and Experience*, 5(5): 407-423, 1993.

[3] F.A. Rabhi *A Parallel Programming Methodology Based on Paradigms.* In Transputer and Occam Developments (18th WoTUG Technical Meeting), P. Nixon (Ed.) IOS Press, 1995: 239-249.

[4] R.W. Floyd. *The Paradigms of Programming.* In ACM Turing Award Lectures: The first Twenty Years: 1966-1985, Ashenhurst and Graham (eds.), ACM Press, N.Y.: 1987, 3-18.

[5] F. Araque, M. Capel, J.M. Mantas, A., Palma. *A Proposal to Improve Reusability in a language based on the Occam-CSP Programming Model.* EUROMICRO Workshop on Parallel Programming, London (UK), 24-27 January 1997. (pending of acceptation)

[6] G. Barret *Occam3 reference manual.* INMOS internal document, 1992.

[7] M. Capel, J.M. Troya, A. Palma. Distributed Active Objects: A Methodological Proposal and Tool for Distributed Programming with Transputer Systems. In EUROMICRO'93 Proceedings, Barcelona (Spain), 6-9 Sept. 1993. *Microprocessing and Microprogramming*, 38: 197-204, 1993.

[8] M. Capel, J.M. Troya. An Object Based Tool and Methodological Approach for Distributed Programming. *Software Concepts and Tools*, 15 (4): 117-195, 1994

[9] H.T. Kung. *Computational Models for Parallel Computers.* In Scientific Applications of Multiprocessors, Elliot and Hoare (Eds.), Prentice-Hall Intnal.: 1989, 1-15.

[10] E. Horowitz, S. Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[11] B. Ibrahim Lopko, G. Padiou. Reusability in the Occam Language. *Structured Programming*, 13: 65-74, 1992.