Managing Processes with Network Objects and Their Translation

Dmitry Arapov, Victor Ivannikov, Alexey Kalinov, Alexey Lastovetsky, Ilya Ledovskih
Institute for System Programming, Russian Academy of Sciences
25, Bolshaya Kommunisticheskaya str., Moscow 109004, Russia
lastov@ispras.ru

Abstract

The mpC language and its supportive portable programming environment are aimed at efficiently-portable modular parallel programming heterogeneous networks of computers (HNCs). Unlike traditional tools used for portable programming HNCs, mpC provides more advanced facilities for process management to support efficient portability. The paper presents the abstraction of network object introduced in the mpC language to manage processes constituting a message-passing program in order to ensure efficient execution of mpC applications on any particular HNC. The main attention is paid to the translation of this high-level mechanism into low-level notions of target message-passing programs.

1. Introduction

Unlike supercomputers - shared-memory multiprocessors (SMPs) and distributed-memory multiprocessors (MPPs), local networks of computers are inherently heterogeneous and comprise diverse computers interconnected via mixed network equipment. In particular, this causes the variety of both performances of computing processor nodes and communication speeds and bandwidths of links interconnecting the nodes.

Nevertheless, for portable high-performance computing on local networks of computers are used mostly the same tools, that are used for portable programming MPPs (and, partially, SMPs), namely, PVM [1], MPI [2] and HPF [3].

To understand how suitable the tools are for local networks, let us estimate how they support efficient, portable, modular, efficiently-portable, easy and reliable programming heterogeneous networks of computers (HNCs).

Efficient programming HNCs means developing such an application that for a particular HNC implements a parallel algorithm utilizing the performance potential of the HNC with sufficient completeness.

Portable programming HNCs means developing such an application that once developed and tested for a particular HNC will run properly on any other HNC without any corrections.

Modular programming HNCs means developing such a parallel program unit that can be separately compiled and correctly used by other programmers when developing their parallel applications without knowing its inside.

Efficiently-portable programming HNCs means developing a portable parallel application able to adapt to peculiarities (in particular, processor performances and communication speeds) of any particular executing HNC to exploit its performance potential with sufficient completeness. (Note, that only efficiently-portable and modular programming HNCs enables the development of parallel packages and libraries for HNCs.)

Easy-in-use and *reliable* programming HNCs means using such a parallel programming model that does not make the development of complex applications for HNCs tedious and error-prone.

PVM, MPI and HPF all support portable programming HNCs.

Unlike PVM, both MPI and HPF additionally support modular programming HNCs.

To understand if MPI and HPF support efficient and efficiently-portable programming HNCs, let us consider some typical parallel algorithms allowing both efficient and efficiently-portable implementation for HNCs and analyze how they can be expressed in MPI and HPF.

Firstly, let us consider a problem of simulating the evolution of a system of stars in a galaxy (or set of galaxies) under the influence of Newtonian gravitational attraction. Let the system consist of a number of large groups of bodies being far away from each other. It is known, that since the magnitude of interaction between bodies falls off rapidly with distance, the effect of a large group of bodies may be approximated by a single equivalent body, if the group of bodies is far enough away from the point at which the effect is being evaluated. So, we can parallelize the problem, and the corresponding message-passing algorithm will use a few processes, each of which updates data characterizing a single group of bodies. Each process holds attributes of all the bodies constituting the corre-

1089-6503/98 $10.00

sponding group as well as masses and centers of gravity of other groups. The attributes characterizing a body include its position, velocity and mass. There is one process, called host-process, that holds all bodies constituting the galaxy and outputs successive states of the galaxy. The scheme of the corresponding message-passing algorithm looks as follows:

```
Initialize a host-process
Initialize a galaxy on the host-process
Initialize a balanced set of processes for the groups
 of the galaxy (one process for each group of the galaxy)
Scatter the groups over the processes
Compute masses of the groups in parallel
Interchange the masses among processes
while(1) {
 Output the current state of the galaxy
   on the host-process
 Compute centers of gravity of the groups in parallel
 Interchange the centers among processes
 Update the groups in parallel
 Gather the groups on the host-process
}
```

Here, "a balanced set of processes" means that the processes are mapped to the executing HNC in such a way to provide the best balance between speeds of processes and speeds of data transfer among the processes minimizing the total running time.

MPI does not allow to create a group of processes based on such properties of the created group as related speeds of processes or speeds of data transfer among processes. The basic way to create a group of processes is to explicitly point out all the processes of the entire ordered set of homogeneous MPI processes running over a HNC, that should join the created group. The only exception is an operation for dividing a single group of processes into n subgroups, each process explicitly specifying which subgroup it wants to join. But even this mechanism does not solve the problem of creating a group of processes based on their relative quantitative characteristics. An MPI process is identified only by its serial number and has no additional properties (attributes) differing it from other MPI processes. Therefore, the programmer, writing the corresponding MPI application, cannot exert influence on the level of the balance among processes of the created group. From MPI's point of view all process groups are equally balanced. This point of view is explainable, if to remember, that MPI was originally developed for portable programming MPPs consisting of identical processors interconnected via very fast network equipment, and any mappings assigning different processes to different processors of a MPP are considered equivalent to each other and optimal. Therefore, there is no problem to start up the MPI application on the MPP in such a way to ensure its efficient execution.

The situation changes drastically for HNCs. Let, for example, a modelled system of bodies consist of 5 groups g1, g2, g3, g4, and g5, comprising 100, 200, 300, 400, and 500 bodies correspondingly, and an executing HNC consist of 5 uniprocessor workstations w1, w2, w3, w4, and w5, relative performances of which are 1, 2, 3, 4, and 5 correspondingly. Simplifying the situation, suppose the workstations to be interconnected with such a network equipment that provide communications, fast enough to neglect communication overheads for the given application. Obviously, that the mapping, assigning g1 to w5, g2 to w4, g3 to w3, g4 to w2, and g5 to w1, will lead to much slower execution of the application, than the mapping, assigning g1 to w1, g2 to w2, g3 to w3, g4 to w4, and g5 to w5. Therefore, to start up the application on a HNC in such a way to ensure its efficient execution, a user should know well about both characteristics of the executing HNC and the inside of the application and solve a non-trivial optimization problem. This problem becomes much more complicated if the executing HNC consists of an rather large number of uni- and multiprocessor computers interconnected with a mixed network equipment or/and if the task of modelling the evolution of the system of bodies is only a part of a larger MPI application.

Thus, when programming HNCs, one cannot use efficiently an MPI module without knowledge of its inside, that is, MPI cannot support both efficient and modular programming HNCs simultaneously.

Moreover, if the number of groups of bodies and the number of bodies in each group are defined only in run time, the user has no information to start up the application in an efficient way. Therefore, in the most general case, MPI does not support efficient programming HNCs and, hence, efficiently-portable programming HNCs.

Like MPI, HPF does not allow to express the above algorithm. The main obstacle is that a homogeneous multiprocessor providing very fast communications among its processors is the only parallel machine visible when programming in HPF. Therefore, a programmer, writing the corresponding HPF application, cannot exert influence on the level of the balance among processes of the target message-passing program. In addition, HPF does not support neither irregular and/or uneven data distribution nor coarse-grained parallelism to express adequately this (rather coarse-grained task than pure data parallel) algorithm. Thus, HPF also does not support efficient (and, hence, efficiently-portable) programming HNCs.

The above problem becomes much more complicated in the case of modelling a system of galaxies each of which in turn consists of a number of groups of bodies. The corresponding message-passing algorithm should deal with an hierarchy of interacting processes, implementing the inher-

ent nested parallelism of this complicated problem, that needs still more sophisticated process management aimed at efficiently-portable running the application on HNCs.

Thus, PVM, MPI and HPF are not suitable for efficiently-portable modular parallel programming HNCs, and new tools taking into account the heterogeneity of the architecture are needed. The first tool of that sort is the research programming system mpC [4-5]. It is based on a C superset providing facilities that allow the user to specify explicitly properties of different groups of processes participating in the execution of different parts of the entire distributed program. The corresponding abstraction is called a network object. In the mpC language, the user can specify topology of and define network objects (in particular, dynamically) as well as distribute data and computations over the network objects. The mpC programming environment uses this information in run time to map the mpC network objects to any underlying HNC in such a way that ensures efficient running of the application on the NHC. The paper is just devoted to implementation of the mechanism of network objects.

2. Outline of the mpC language

In mpC, the notion of *computing space* is defined as a set of typed virtual processors of different performance connected with links of different length, accessible to the user for management via creation and deallocation of so-called *network objects*. A network object is a region of the computing space which can be used to compute expressions and execute statements. Allocating network objects in the computing space and discarding them is performed in similar fashion to allocating data objects in storage and discarding them.

For example, the user can write in mpC the following

```
/*1*/  nettype HeteroNet(n, p[n]) {
/*2*/    coord I=n;
/*3*/    node { I>=0: p[I]; };
/*4*/  };
/*5*/  ...
/*6*/      {
/*7*/        repl int m, q[N];
/*8*/        /*Compute m,q[0],...,q[m-1]*/
/*9*/        {
/*10*/         net HeteroNet(m,q) r;
/*11*/         ...
/*12*/       }
/*13*/     }
```

to define automatic network object `r` consisting of `m` virtual processors, the relative performance of the `i`-th virtual processor being characterized by the value of `q[i]`.

Here, lines 1-4 declare network topology `HeteroNet` parametrized with integer parameter `n` and vector parameter `p` consisting of `n` integers. Line 2 is a coordinate declaration declaring the coordinate system to which virtual processors are related. It introduces coordinate variable `I` ranging from 0 to `n-1`. Line 3 is a node declaration. It relates virtual processors to the coordinate system and declares their types and performances. It stands for the predicate ***for all*** `I<n` ***if*** `I>=0` ***then*** *a virtual processor, whose relative performance is specified by the value of* `p[I]`*, is related to the point with the coordinate* `[I]`.

Line 7 defines variable `m` and array `q` both replicated over the entire computing space (any network object is considered a region of the entire computing space). By definition, data object *distributed* over a region of the computing space comprises a set of components of any one type so that each virtual processor of the region holds one component. By definition, a distributed data object is *replicated* if all its components is equal to each other.

Conceptually, creation of a new network object is initiated by a virtual processor of a network object already created. This virtual processor is called a *parent* of the created network object. The parent belongs to the created network object. In our case, the parent of network object `r` is the so-called virtual *host-processor* - the only virtual processor defined from the beginning of program execution till program termination.

Suppose we to model the evolution of `m` groups of bodies under the influence of Newtonian gravitational attraction, and our parallel application uses a virtual processor to update a single group. Suppose also `q[i]` to be equal to the square of the number of bodies in the `i`-th group. Then, line 10 defines network object `r`, executing most of computations and communications, in such a way, that it consists of `m` virtual processors, and the relative performance of each processor is characterized by the volume of computations to update the group which it computes. So, the more powerful is the virtual processor, the larger group of bodies it computes. The mpC programming environment bases on this information as well as on the information about the topology of an underlying HNC to map the virtual processors into the processes, running on this HNC and representing the entire computing space, in the most appropriate way. Since it does it in run time, the user does not need to recompile the mpC code to port it to other HNCs.

So, unlike MPI and HPF, supporting efficiently-portable modular programming MPPs, mpC also supports efficiently-portable modular programming HNCs, including MPPs as particular cases.

3. Translation of network definitions

The mpC compiler translates a source mpC file into a target ANSI C file with calls to functions of the run-time

support system. It uses the SPMD model of target code, when all processes constituting the target mpC program run identical code.

All processes constituting the target program are divided into 2 groups - a special process, called *dispatcher,* playing the role of the manager of the computing space, and common processes, called *nodes,* playing the role of virtual processors of the computing space. The dispatcher works as a server. It receives requests from nodes and sends them commands.

In the target program, every network of the source mpC program is represented by a set of nodes called *region.* At any time of the target program running, any node is either free or hired in one or several regions. Hiring nodes in created regions and dismissing them are responsibility of the dispatcher. The only exception is the pre-hired host-node representing the mpC pre-defined virtual host-processor. Thus, just after initialization, the computing space is represented by the host and a set of temporarily free (unemployed) nodes.

The main problem in managing processes is hiring them to network regions and dismissing them. A solution of this problem establishes the whole structure of the target code and forms requirements for functions of the run-time support system.

To create a network region, its parent node computes, if necessary, parameters of the corresponding network topology and sends a creation request to the dispatcher. The request contains full topological information about the created region including the number of nodes and their relative performances. On the other hand, the dispatcher keeps information about the topology of the target network of computers including the number of actual processors, their relative performances and the mapping of nodes onto the actual processors. Based on the topological information, the dispatcher selects the set of free nodes, which is most appropriative to be hired in the created network region. (More detailed description how dispatcher does it, may be found in [8].) After that, it sends to every free node a message saying whether the node is hired in the created region or not.

To deallocate a network region, its parent node sends a message to the dispatcher. Note, that the parent node leaves hired in the parent-network region of the deallocated region. The rest of members of the deallocated network region become free and begin waiting for commands from the dispatcher.

Any node can detect its hired/free status. It is hired if a call to function `MPC_Is_busy` returns 1. If such a call returns 0, the node is free.

Any node can detect if it is hired in some particular region or not. A region is accessed via its descriptor. If the descriptor `rd` corresponds to the region, then a node belongs to the region if and only if the function call `MPC_Is_member(&rd)` returns 1. In this case, descriptor `rd` allows the node to obtain comprehensive information about the region as well as identify itself in the region. The region descriptor has type `MPC_Net` and holds the following data:

- topological data associated with the region, such as the number of coordinates, an integer array containing actual topological arguments (if any) and the number of elements in this array, pointers to the corresponding topological functions;
- the number of nodes in the region;
- the linear number of the node in the region;
- an integer array containing coordinates of the given node in the corresponding network;
- some additional and /or redundant information aimed at optimization of computations and communications.

When a free node is hired in a network region, the dispatcher must let the node know, in which region it is hired, that is, must specify the descriptor of that region. The simplest way - to pass the pointer to the region descriptor from the parent node through the dispatcher to the free node, is senseless for distributed memory systems not having common address space. Therefore, in addition to the region descriptor, something else is needed to identify the created region in a unique fashion. The additional identifier must have the same value on both the parent and the free node and be passable from the parent node through the dispatcher to the free node.

In a source mpC program, a network is denoted by its name, being an ordinary identifier and not having to have file scope. Therefore, a network name can not serve as a unique network identifier even within a file. One could enumerate all networks declared in the file and use the number of a network as an identifier unique within the file. However, such an identifier being unique within a file can not be used as a unique identifier within the whole program that may consist of several files. Nevertheless, one can use it without collisions when creating network regions, if during network-region creation all participating nodes execute the target code located in the same file. Our compiler just enumerates networks defined in a file and uses their numbers as network identifiers in target code when creating the corresponding network regions. It does ensure that during the creation of a network region all involved nodes execute the target code located in the same file.

Creating a network region involves its parent node, all free nodes and the dispatcher. The parent node calls to function

```
int MPC_Net_create(MPC_Name name,
                   MPC_Net* net);
```

where `name` contains the unique number of the created network in the file, and `net` points to the corresponding

region descriptor. The function computes all topological information and sends a creation request to the dispatcher.

Meantime, free nodes are waiting for commands from the dispatcher at so-called *waiting point* calling the function

```
int MPC_Offer(const MPC_Name* names,
              MPC_Net** nets_voted,
              int voted_count);
```

where `names` is an array of numbers of all networks the creation of which are expected at the waiting point, `nets_voted` points to an array of pointers to descriptors of the regions the creation of which are expected at the waiting point, `voted_count` contains the number of elements in array `names`.

The correspondence between the network numbers and region descriptors is established in the following way. If a free node receives from the dispatcher a message saying that it is hired in a network the number of which is equal to `names[i]`, then the node is hired in the network region the descriptor of which is pointed by `nets_voted[i]`.

A free node leaves the waiting function `MPC_Offer` either after it becomes hired in a network region or after the dispatcher sends to all free nodes the command to leave the current waiting point.

4. Structure of target code for mpC block

In general, target code for an mpC block with network definitions has two waiting points. In the first, called *creating waiting point*, free nodes are waiting for commands on region creations. In the second, called *deallocating waiting* point, they are waiting for commands on region deallocation. In general, free nodes participate not only in creation/deallocation of regions for networks defined in the mpC block, but also in overall computations (that is, in computations distributed over the entire computing space) and/or in creation/deallocation of regions for networks defined in nested blocks. Let us call the first mpC statement in the block involving all free nodes in its execution a *waiting-point break statement*.

Then, in the most general case, the compiler generates target code of the following structure:

```
{
  declarations
  {
   if(!MPC_Is_busy()) {
    target code executed by free nodes
    to create regions for networks
    defined in source mpC block
   }
   if(MPC_Is_busy()) {
    target code executed by hired
    nodes to create regions for
    networks defined in source mpC-block
    and
    target code for mpC statements
    before waiting-point break statement
   }
   epilogue of waiting point
  }
  target code for mpC statements, starting
  from waiting-point break statement
  {
    target code executed by hired nodes
    to deallocate regions for networks
    defined in source mpC block

    label of deallocating waiting point:
    if(!MPC_Is_busy()) {
      target code executed by free nodes
      to deallocate regions for networks
      defined in source mpC block
    }
    epilogue of waiting point
  }
}
```

If the source mpC block does not contain a waiting-point break statement (that is, overall statements and nested blocks with network definitions or overall statements), then creating and deallocating waiting points can be merged. Let us call such a waiting point *shared waiting point*. Target code for the mpC block with a shared waiting point looks as follows:

```
{
  declarations
  {
    label of shared waiting point:
    if(!MPC_Is_busy()) {
      target code executed by free nodes
      to create and deallocate regions for
      networks defined in source mpC block
    }
    if(MPC_Is_busy()) {
      target code executed by hired nodes
      to create and deallocate regions for
      networks defined in source mpC block
      and
      target code for statements
      of source mpC block
    }
    epilogue of waiting point
  }
}
```

To ensure that during the creation of a network region all involved nodes execute target code located in the same file, the compiler put a global barrier into the epilogue of waiting point.

The coordinated arrival of nodes to the epilogue of waiting point is ensured by the following scenario:

- the host makes sure that all other hired nodes, which

might send a creation/deallocation request expected in the waiting point, have already reached the epilogue;

- after that, the host sends a message, saying that any creation/deallocation request expected in the waiting point will not appear yet, to the dispatcher;

- after receiving the message the dispatcher sends all free nodes a command ordering to leave the waiting point;

- after receiving the command each free node leaves the waiting function and reach the epilogue.

5. Process management in details

To introduce the process management in more details, let us consider the following mpC file:

```
/*1 */nettype T(m) { coord I=m; };
/*2 */void [*]f(int [host] hn) {
/*3 */  net T(2) n;
/*4 */  repl in;
/*5 */  in=hn;
/*6 */  {
/*7 */    net T(in) [n] nn;
/*8 */    .../* declarations*/
/*9 */    .../*statements without a waiting-
/*10*/         point break statement*/
/*11*/  }
/*12*/}
```

Line 1 introduces topology T with parameter m. It describes networks consisting of m virtual processors with the integer coordinate variable I ranging from 0 to m-1.

Line 3 defines network n consisting of two virtual processors.

Line 4 defines integer variable in replicated over the entire computing space.

Line 5 broadcasts the value of variable hn from the virtual host-processor over the entire computing space. The statement is executed by the entire computing space. Therefore, it is a waiting-point break statement for the function body.

Line 7 defines network nn. The network nn is a distributed network. In general, mpC allows to define not only a single network but also a set of single networks by means of defining so-called *distributed network*. A definition of a distributed network specifies the type of the network and its parent network. Such a definition may be considered as a distributed over the parent network definition of a single network of the specified type. The parent network of a distributed network can also be distributed. But in any case, a distributed network is a set of single networks of the same type. The number of single networks in this set is equal to the number of virtual processors in the parent network each of the virtual processors of the parent network being a parent of a single network of the set.

There are not facilities to specify a single network belonging to a distributed network in mpC. Therefore, whenever one specifies a subnetwork of a distributed network, he means a set of subnetworks of the single networks constituting the distributed network. Similarly, if one specifies a single processor of a distributed network, he means a set of single processors of the single networks constituting the distributed network. Any computation on a distributed network is divided into independent computations on the single networks constituting the distributed network.

So, network nn, distributed over its parent network n, divides into a set of two single networks the type of which is defined completely only in run time.

There will be all three kinds of waiting points in target code for function f. The function body, where network n is defined, contains a waiting-point break statement. Therefore, target code for the function body will contain both creating and deallocating waiting points. The nested block (lines 6-11), where network nn is defined, does not contain a waiting-point break statement. Therefore, target code for the nested block will contain a shared waiting point.

The following target code

```
/*1 */void f() {
/*2 */ int MPC_Net_n_6_coord[1];
/*3 */ MPC_Parameters
/*4 */       MPC_Net_n_6_params[1]={2};
/*5 */ MPC_Net MPC_Net_n_6={...
/*6 */      /* initialization list */};
/*7 */ int in;
/*8 */ {
/*9 */  if(!MPC_Is_busy()){
/*10*/   MPC_Name MPC_names[1]={6};
/*11*/   MPC_Net* MPC_nets[1];
/*12*/   MPC_nets[0]=&MPC_Net_n_6;
/*13*/   MPC_Offer(MPC_names,MPC_nets,1);
/*14*/  }
/*15*/  if(MPC_Is_busy()){
/*16*/   if(MPC_Is_member(&MPC_Net_host)){
/*17*/    MPC_Net_create(6,&MPC_Net_n_6);
/*18*/   }
/*19*/   if(MPC_Is_host()){
/*20*/    MPC_Host_out();
/*21*/   }
/*22*/  }
/*23*/  MPC_Waiting_point_end();
/*24*/ }
/*25*/ /* implementation of in=ih. */
/*26*/ {
/*27*/  int MPC_Net_nn_7_coord[1];
/*28*/  MPC_Parameters
/*29*/        MPC_Net_nn_7_params[1];
/*30*/  MPC_Net MPC_Net_nn_7={...
/*31*/       /*initialization list*/};
/*32*/  /*target code for declarations of
/*33*/    the source nested block*/
/*34*/  {
```

```
/*35*/   MPC_waiting_point_2:
/*36*/   if(!MPC_Is_busy()){
/*37*/    MPC_Name MPC_names[1]={7};
/*38*/    MPC_Net* MPC_nets[1];
/*39*/    MPC_nets[0]=&MPC_Net_nn_7;
/*40*/    MPC_Offer(MPC_names,MPC_nets,1);
/*41*/   }
/*42*/   if(MPC_Is_busy()){
/*43*/    if(MPC_Is_member(&MPC_Net_n_6)){
/*44*/     MPC_Net_nn_7.count=1;
/*45*/     MPC_Net_nn_7.params=
/*46*/              MPC_Net_nn_7_params;
/*47*/     {
/*48*/      MPC_Net_nn_7_params[0]=in;
/*49*/     }
/*50*/     MPC_Net_create(7,&MPC_Net_nn_7);
/*51*/    }
/*52*/    /*target code for statements of
/*53*/      the source nested block*/
/*54*/    if(MPC_Is_member(&MPC_Net_nn_7)){
/*55*/     MPC_Net_free(&MPC_Net_nn_7);
/*56*/     if(!MPC_Is_busy())
/*57*/      goto MPC_waiting_point_2;
/*58*/    }
/*59*/    if(MPC_Is_member(&MPC_Net_n_6)){
/*60*/     MPC_Local_barrier(&MPC_Net_n_6);
/*61*/    }
/*62*/    if(MPC_Is_host()){
/*63*/     MPC_Host_out();
/*64*/    }
/*65*/   }
/*66*/   MPC_Waiting_point_end();
/*67*/  }
/*68*/  if(MPC_Is_member(&MPC_Net_n_6)){
/*69*/   MPC_Net_free(&MPC_Net_n_6);
/*70*/   if(!MPC_Is_busy())
/*71*/    goto MPC_reconfig_point_1;
/*72*/  }
/*73*/  if(MPC_Is_host()){
/*74*/   MPC_Host_out();
/*75*/  }
/*76*/  MPC_reconfig_point_1:
/*77*/  if(!MPC_Is_busy()){
/*78*/   MPC_Offer(NULL,NULL,0);
/*79*/  }
/*80*/  MPC_Waiting_point_end();
/*81*/ }
/*82*/}
```

is generated by the mpC compiler for the above mpC function `f`. Lines 1-25 and lines 68-82 are generated for the function body, and lines 26-67 are generated for the nested block.

Lines 2-24 of the target code are related to the creating waiting point. The network `n` has obtained number 6 as an identifier unique in the file, and the corresponding network region is accessible via descriptor `MPC_Net_n_6` (line 5). Line 2 defines the one-element array `MPC_Net_n_6_coord` to hold the coordinates of nodes of the region. Lines 3-4 define and initialize the one-element array `MPC_Net_n_6_params` in such a way that its only element holds integer value 2 as an argument of topology `T` establishing the type of network `n` (namely, the network type `T(2)`). Lines 5-6 define the region descriptor `MPC_Net_n_6` and initialize all such its members, values of which can be computed in compile time.

The target code in lines 10-13 is executed by all free nodes to create the region represented the network `n`. Line 10 defines and initializes the one-element array `MPC_names` containing the number of the network the creation of which is expected at the first waiting point. Line 11 defines the one-element array `MPC_nets` to hold a pointer to the region descriptor the creation of which is expected at the first waiting point, and line 12 assigns the proper value to its only element. Line 13 calls to the waiting function `MPC_Offer`. A free node leaves the function either after it becomes hired in region `MPC_Net_n_6`, or after the dispatcher sends to all free nodes the command to leave this waiting point.

The target code in lines 16-24 is executed by all hired nodes to create the region for network `n` and to reach coordinately the epilogue of the creating waiting point.

Lines 16-18 call to the function `MPC_Net_create` on the host to form the corresponding creation request and to send it to the dispatcher. The host is accessible via descriptor `MPC_Net_host`. Any node is detect itself as the host if the function call `MPC_Is_member(&MPC_Net_host)` or the function call `MPC_Is_host()` return 1 on the node.

Lines 19-21 call to the function `MPC_Host_out` on the host to send the dispatcher a message saying that all free nodes must leave the waiting point. Since in our example the host is the only node, that can send a creation request expected in the waiting point, it knows that all creation requests expected in the waiting point have already been sent, and it may send the message to the dispatcher.

The statement in line 23 calls to the function `MPC_Waiting_point_end`. It is an epilogue of the first waiting point. The call provides a global barrier synchronization and does not let any node to continue until all nodes constituting the entire computing space reach it.

The target code for the nested block (lines 26-67) is related to the shared waiting point. The network `nn` has obtained number 7 as an unique identifier in the file, and the corresponding network region is accessible via descriptor `MPC_Net_nn_7` (line 30).

Line 27 defines the one-element array `MPC_Net_nn_7_coord` to hold the coordinates of nodes of the region. Lines 28-29 define the one-element array `MPC_Net_nn_7_params` to hold an argument of topology `T` establishing the type of network `nn`. Lines 30-

31 define the region descriptor MPC_Net_nn_7 and initialize all such its members, values of which can be computed in compile time.

The target code in lines 37-40 is similar to the target code in lines 10-13 and executed by all free nodes to create the region represented the network nn.

The target code in lines 43-65 is executed by all hired nodes to create and deallocate the region for network nn, to execute statements of the source mpC nested block, and to reach coordinately the epilogue of the shared waiting point.

Lines 44-50 are executed by two nodes constituting region MPC_Net_nn_6 in parallel to create two regions representing the distributed network nn. Lines 44-49 compute some attributes of these regions, allowing to establish the type of network nn, and store them in the corresponding members of the region descriptor MPC_Net_nn_7. Line 50 calls to the function MPC_Net_create to form two corresponding creation requests and to send them to the dispatcher.

Lines 55-57 are executed on the regions representing network nn to deallocate them. Line 55 calls to the function MPC_Net_free. The function provides a local barrier synchronization over the deallocated regions. After all nodes constituting these regions reach the local barrier, each of two nodes constituting their parent region (that is, region MPC_Net_nn_6) send a message to the dispatcher. These two nodes remain to be hired in region MPC_Net_nn_6. Meantime, other members of the distributed network region become free and jump to the label MPC_waiting_point_2 of the shared waiting point (line 57). They begin executing the free-node code (lines 37-40) and, eventually, join other free nodes calling the waiting function in line 40.

Lines 59-64 ensure that all nodes reach the epilogue of the shared waiting point coordinately. Since the host is not the only node that can send a request expected in the waiting point, it can not pass over the local barrier in line 60 and call to function MPC_Host_out in line 63 to send the dispatcher a message saying that all free nodes must leave the waiting point, until all other nodes able to send a creation/deallocation request reach the local barrier.

As a result, all nodes call to the epilogue function MPC_Waiting_point_end (line 66) coordinately.

The rest of the target code generated for the function body (lines 68-82) is related to the deallocating waiting point.

Lines 68-75 are executed by hired nodes to deallocate the region representing the network n and to ensure that all nodes reach the epilogue of the deallocating waiting point coordinately. The node, that becomes free after the call to function MPC_Net_free in line 69, jumps to the label MPC_reconfig_point_1 of the deallocation waiting point (line 71) and calls to the waiting function (line 78). Since the host is the only node that can send a deallocation request expected in the waiting point, it does not need to synchronize its work with some other hired nodes and can call to the function MPC_Host_out to send the dispatcher a message saying that free nodes must leave the deallocation waiting point.

Lines 77-79 ensure that all free nodes will receive in time the command from the dispatcher to leave the waiting point and will reach the epilogue function (line 80) coordinately with the hired nodes.

6. Conclusion

The paper has presented the abstraction of network object introduced in the mpC language to manage processes constituting a message-passing program in order to ensure an efficient execution of mpC applications on any particular HNC. The main attention has been paid to the translation of this high-level mechanism into low-level notions of the target message-passing program. The presented algorithm has overcome 2-year intensive testing and been incorporated into the mpC programming environment, widely used for efficiently-portable modular parallel programming local networks of diverse workstations, servers and PCs.

References

[1] A.Geist, A.Beguelin, J.Dongarra, W.Jlang, R.Manchek, V.Sunderam, *PVM: Parallel Virtual Machine, Users' Guide and Tutorial for Networked Parallel Computing,* MIT Press, Cambridge, MA, 1994.

[2] Message Passing Interface Forum, *MPI: A Message-passing Interface Standard, version 1.1,* June 1995.

[3] High Performance Fortran Forum, *High Performance Fortran Language Specification, version 2.0.* Rice University, Houston TX, 1997.

[4] A. Lastovetsky, "mpC - a Multi-Paradigm Programming Language for Massively Parallel Computers", *ACM SIGPLAN Notices, 31(2),* February 1996, pp.13-20.

[5] D.Arapov, A.Kalinov, A.Lastovetsky, I.Ledovskih, and T.Lewis, "A Programming Environment for Heterogeneous Distributed Memory Machines", *Proceedings of the Sixth Heterogeneous Computing Workshop (HCW'97),* IEEE CS Press, Geneva, Switzerland, April 1, 1997, pp.32-45.