Resource Management in the mpC Programming Environment

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

Abstract

The mpC language was developed to write efficient and portable programs for wide range of distributed memory machines. It supports both task and data parallelism, allows both static and dynamic process and communication structures, enables optimizations aimed at both communication and computation, and supports modular parallel programming and the development of a library of parallel programs. The language is an ANSI C superset based on the notion of network comprising processor nodes of different types and performances, connected with links of different bandwidths. The user can describe a network topology, create and discard networks, distribute data and computations over networks. The mpC programming environment uses the topological information in run time to ensure the efficient execution of the application.The paper describes the implementation of managing networks in the mpC programming environment.

1. Introduction

Programming for distributed memory machines (DMMs) is based mostly on message-passing function extensions of C or Fortran, such as PVM [1] and MPI [2]. It is tedious and error-prone to program in a message-passing language, because in fact such a language is of assembly level for DMMs. Therefore, high-level languages that facilitate parallel programming are developed.

Currently, any high-level programming language for DMMs can be put in one of two classes depending on the parallel programming paradigm - task parallelism or data parallelism - underlying it. Task parallel [3-4] and data parallel [5-11] programming languages allow the user to implement different classes of parallel algorithms. But efficient implementation of many problems needs parallel algorithms that can not be implemented in pure data parallel or task parallel styles. Few examples of such algorithms can be found in [12].

The mpC language is an ANSI C superset developed to support both task and data parallelism, to allow both static and dynamic process and communication structures, to enable optimizations aimed at both communication and computation, and to support modular parallel programming and the development of a library of parallel programs. It is based on the notion of network comprising processor nodes of different types and performances connected with links of different bandwidths. It has facilities for describing network topology, for creating and discarding networks, for distribution of data and computations over networks.

All programming environments for DMMs which we know of have one common weakness. Namely, when developing a parallel program, either the user has no facilities to describe the virtual parallel system executing the program, or such facilities are too poor to specify an efficient distribution of computations and communications over the target DMM. Even MPI's topological facilities have turned out insufficient to solve the problem. So, to ensure the efficient execution of the program on a particular DMM, the user must use facilities which are external to the program, such as boot schemes and application schemes [13]. If the user is familiar with both the topology of target DMM and the topology of the application, then, by using such configurational files, he can map the processes which constitute the program onto processors which make up DMM, to provide the most efficient execution of the program. But if the application topology is defined in run time (that is, if it depends on input data), it won't be successful.

The mpC language allows the user to specify an application topology (in particular, dynamically), then the mpC programming environment uses this information to map processes onto processors of the target machine resulting in the efficient execution of the application. Since it does it in run time, you need not recompile the mpC program, when transferring it to another DMM. Therefore, mpC and its programming environment allow to write applications which once compiled will run efficiently on any DMM.

DMM's topology is detected automatically dependent on execution of a special program.

This paper outlines the mpC features concerning networks and presents the implementation of network management in the mpC programming environment. Details of the language are presented elsewhere [14-16].

2. Resource management in mpC

In mpC, the notion of *computing space* is defined as a set of typed virtual processors of different performance connected with links of different bandwidth accessible to the user for management. There are three processor types: `memory`, `scalar`, and `vector`. A processor of the `memory` type can rather store data than operate on them. A processor of the `vector` type can perform vector operations efficiently. Finally, most common processors are of the `scalar` type. Besides, a processor has additional attribute characterizing its relative performance. A directed *link* connecting two virtual processors is a one-way channel for transferring data from source processor to the processor of destination.

The basic notion of the mpC language is *network object* or simply *network.* Network comprises processor nodes of different types and performances connected with links of different bandwidths. Network 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 a similar fashion as allocating data objects in the storage and discarding them. Conceptually, creation of new network is initiated by a processor of a defined network. This processor is called a *parent* of the created network. The parent belongs to the created network. The only processor defined from the beginning of program execution till program termination is the pre-defined *host-processor* of the `scalar` type.

Every network object declared in mpC program has a type. The type specifies the number and types and performances of processors, links between the processors and their lengths characterizing bandwidths, as well as separates the parent. For example, the type declaration

```
/* 1 */  nettype Rectangle {
/* 2 */    coord I=4;
/* 3 */    node {
/* 4 */      I<2 : fast scalar;
/* 5 */      I>=2: slow scalar;
/* 6 */    }
/* 7 */    link {
/* 8 */      I>0:  [I]<->[I-1];
/* 9 */      I==0: [I]<->[3];
/* 10 */   }
/* 11 */   parent [0];
/* 12 */ }
```

introduces the network type named `Rectangle` that corresponds to networks consisting of four processors of the `scalar` type and different performances interconnected with undirected links of normal length in rectangular structure.

In this example, line 1 is a header of the network-type declaration. It introduces the name of the network type.

Line 2 is a coordinate declaration declaring the coordinate system to which processors are related. It introduces the integer coordinate variable `I` ranging from 0 to 3.

Lines 3-6 are a node declaration. It relates processors to the coordinate system and declares their types and performances. The line 4 stands for the predicate ***for all*** *`I<4`* ***if*** *`I<2`* ***then*** *fast processor of the `scalar` type is related to the point with the coordinate `[I]`*. The line 5 stands for the predicate ***for all*** *`I<4`* ***if*** *`I>=2`* ***then*** *slow processor of the `scalar` type is related to the point with the coordinate `[I]`*. The performance specifiers `fast` and `slow` specify relative performances of processor nodes of the same type. For any network of this type, this information allows the compiler to associate a weight with each processor of the network normalizing it in relation to the weight of the parent processor. Note, that the host-processor is always of the `scalar` type and normal performance.

Lines 7-10 are a link declaration. It specifies links between processors. Line 8 stands for the predicate ***for all*** *`I<4`* ***if*** *`I>0`* ***then*** *there exists undirected link of normal length connecting processors with coordinates `[I]` and `[I-1]`,* and line 9 stands for the predicate ***for all*** *`I<4`* ***if*** *`I==0`* ***then*** *there exists undirected link of normal length connecting processors with coordinates `[I]` and `[3]`.* Note, that if a link between two processors is not specified explicitly, it is meant that there is a link whose length is longest for this network.

Line 11 is a parent declaration. It specifies that parent processor has the coordinate [0].

With the network type declaration, one can declare a network object identifier of this type. For example, the declaration

```
    net Rectangle r1, r2;
```

introduces two identifiers r1, r2 of network objects. Here, `net  Rectangle` is a full name of the corresponding type.

Besides a network type, one can declare a parametrized family of network types called *topology* or *generic network type*. For example, the declaration

```
/* 1 */  nettype Ring(n, power[n]) {
/* 2 */    coord I=n;
/* 3 */    node {
/* 4 */      I>=0: fast*power[I] scalar;
/* 5 */    }
/* 6 */    link {
/* 7 */      I>0:  [I]<->[I-1];
```

```
/* 8 */       I==0: [I]<->[n-1];
/* 9 */     }
/* 10 */   parent [0];
/* 11 */ }
```

introduces the topology named `Ring` that corresponds to networks consisting of `n` processors of the `scalar` type interconnected with undirected links of normal length in a ring structure. The header (line 1) introduces parameters of the topology `Ring`, namely, the integer parameter `n` and the vector parameter `power` consisting of `n` integers. (So, these two topological parameters totally include `n+1` integers.) Correspondingly, the coordinate variable `I` ranges from `0` to `n-1`, line 4 stands for the predicate ***for all*** *I<n* ***if*** *I>=0* ***then*** *fast processor of the scalar type, whose relative performance is specified by the value of power[I], is related to the point with the coordinate [I]*, and so on. Here, the performance specifier `fast*power[i]` includes the so-called power specifier `*power[I]`. In general, the value of the expression in a power specifier shall be positive integer. Any operand in the expression shall consist only of coordinate variables, constants and generic parameters (if any). If the value of the expression is equal to 1, the power specifier may be omitted. It is meant that in the framework of the same network-type declaration any performance specifier with the `fast` keyword specifies more powerful processor than a performance specifier with the `slow` keyword. It is also meant that the greater value of the expression in a power specifier, the more performance is specified.

With the topology declaration, one can declare a network object identifier of the proper type. For example, the declarations

```
net Ring(4,{1,2,3,4}) r;
```

introduces the network type `Ring(4,{1,2,3,4})` as an instance of the topology `Ring` as well as the identifier `r` of the network object of this type. Here, `4` and `{1,2,3,4}` are actual topological arguments.

The notion of *distributed data object* is introduced in the spirit of C* [9] and Dataparallel C [10]. Namely, data object distributed over a region of the computing space comprises a set of components of any one type so that every processor of the region holds one component. For example, the declarations

```
net Rectangle r2;
int [*]Derror, [r2]Da[10];
repl [*]Di;
```

declare `Derror` as an integer data object distributed over the entire computing space, `Da` as an array of 10 `int`s distributed over the network `r2`, and `Di` as an integer variable replicated over the entire computing space. By definition, a distributed object is *replicated* if all its components is equal to each other.

The notion of *distributed value* is introduced similarly.

An instance of topology can be obtained not only statically but also dynamically. For example, the fragment

```
repl m, n[100];
/* Computing m, n[0],...,n[m-1] */
{
  net Ring(m,p) rr;
  ...
}
```

introduces the identifier `rr` of the network object the type of which is defined completely only in run time.

A network object has a computing space duration that determines its lifetime. There are two computing space durations: static, and automatic. A network object declared with *static* computing space duration is created only once, conceptually just before its first usage. It exists till termination of the entire program. A new instance of a network object declared with *automatic* computing space duration is created on each entry into the block in which it is declared. The network object is discarded when execution of the block ends.

A declaration of a network object identifier specifies the scope and the linkage of the identifier and the computing space duration of the network object almost under the same rules that are used for specification of storage duration of data objects and scopes and linkages of their identifiers. Network object declaration that also causes computing space to be reserved for the network object named by an identifier is a *network object definition*. The parent of all above network objects is the host-processor. The following example shows how one can specify another parent:

```
net Ring(3,p1) r3;
net Ring(4,p2) [r3:I==1]r4;
```

Here, the network `r3` has the host-processor as its parent, meantime the network `r4` has the processor of the network `r3` with the coordinate `[1]` as its parent.

To support nested parallelism, mpC allows one to define not only a single network but also a set of single networks by means of defining so-called *distributed network object*. For example, the declarations

```
net Ring(5,p3) r5;
net Ring(3,p4) [r5] r3;
```

declare the network `r3` distributed over the network `r5`. In fact, `r3` is a set of `5` networks each of which contains `3` processors connected in ring. Five parent processors of these 5 networks are also connected in a ring constituting the network `r5`.

A new network object can be allocated not only in an unused computing space but also in a region of the computing space that already holds another network object. It can be done by explicit or implicit definition of a *subnetwork* of existing network object. For example, the declarations

```
net Ring(5,p5) r5;
subnet [r5:I<3] line3;
```

define explicitly the subnetwork `line3` of the network `r5`.

An expression can be evaluated and a statement can be executed by the host-processor, by a single processor of a network, by a network or a subnetwork, by a set of networks and/or subnetworks, or by the entire computing space. In the latter three cases, they are *distributed*. The value of a distributed expression may be also distributed. A set of distributed statements includes the sequential C statements extended with distributed data as well as the special parallel statements `fan`, `par` and `pipe`. If evaluation of expression or execution of statement involves the entire computing space, such an expression or a statement is *overall*.

A special type of a distributed expression is an *asynchronous* expression. Substantially, evaluation of an asynchronous expression needs neither data exchange between processors of the evaluating region of the computing space nor synchronization of the processors during its evaluation. The property of asynchronity of an expression is determined by the property of asynchronity of operators forming the expression. Most of operators of the mpC language are asynchronous in the sense that either both operands and the result belong to the same processor, or they both are distributed through the same region of the computing space, and the distributed operator is divided into a set of independent undistributed operators each of which is performed on corresponding components of the operands. If an expression is built only from such operators, and all they are distributed through the same region of the computing space, then the entire expression will be asynchronous.

The notion of *asynchronous* statement is defined similarly. In particular, if all expressions and substatements of a sequential statement are asynchronous and distributed through the same region of the computing space, then the statement is asynchronous. In this case, the distributed statement is divided into a set of independent undistributed statements each of which is executed by the corresponding processor using the corresponding data components.

Execution of an mpC program begins from a call to the function named `main` on the entire computing space.

The following simple mpC program computes the dot product of two vectors.

```
/*Line 1 */  nettype Star(n) {
/*Line 2 */    coord I=n;
/*Line 3 */    node { default: scalar;};
/*Line 4 */    link { I>0: [0]<->[i];};
/*Line 5 */    parent [0];
/*Line 6 */  };
/*Line 7 */  #define N 100
/*Line 8 */  void [*]main()
/*Line 9 */  { double [host]x[N];
/*Line 10*/    double [host]y[N];
/*Line 11*/    double [host]z;
/*Line 12*/    double sqrt();
/*Line 13*/    /*Input of x and y */
/*Line 14*/    { net Star(N) s;
/*Line 16*/      double [s]dx, [s]dy, [s]dz;
/*Line 17*/      dx=x[];
/*Line 18*/      dy=y[];
/*Line 19*/      dz=dx*dy;
/*Line 20*/      z=[host](dz[+]);
/*Line 21*/      z=([host]sqrt)(z);
/*Line 22*/    }
/*Line 23*/    /* Output of z */
/*Line 24*/  }
```

The program includes 2 functions - `main` defined here and the library function `sqrt`. Lines 8-24 contain the `main` definition. Lines 9-11 contain the definitions of arrays `x`, `y` and variable `z` all belonging to the host-processor. Line 12 contains the declaration of the function identifier `sqrt`. In general, mpC allows 3 kinds of functions. Here, functions of two kinds are used: `main` is a *basic* function, and `sqrt` is a *nodal* function.

A call to *basic function* is an overall expression. Its arguments (if any) shall either belong to the host-processor or be replicated over the entire computing space, and its value (if any) shall belong to the host-processor. In contrast to other kinds of function, it can define network objects. In line 8, construct `[*]`, placed just before the function identifier, specifies that `main` is an identifier of basic function.

Nodal function can be executed completely on any single processor. Only local data objects of the executing processor may be defined in such a function. In addition, the corresponding component of an externally-defined distributed data object can be used in the function. A declaration of nodal function (e.g., in line 12) does not need any additional specifiers.

Line 14 defines the automatic network `s` with the host-processor as a parent. Line 16 defines three automatic variables `dx`, `dy`, and `dz` all distributed over `s`. All the statements in lines 17-20 are distributed over `s`, but only the statement in line 19 is asynchronous. Line 21 contains the undistributed statement executed by the host-processor. It performs, in particular, calling function `sqrt` on the host-processor.

Line 17 contains the unusual unary postfix operator `[]`. The point is that, strictly speaking, mpC is a superset of the vector extension of ANSI C named the C[] language [17], where the notion of *vector,* defined as an ordered sequence of values of any one type, is introduced. In contrast to an array, a vector is not a data object but just a new kind of value. In particular, the value of an array is a vector. The operator `[]` was introduced to support access to an array as a whole. It has an operand of the type "array of *type*" and blocks conversion of the operand to pointer.

So, `x[]` designates array `x` as a whole, and `dx=x[]`

scatters the elements of `x` to the components of `dx`.

Similarly, the statement in line 18 scatters the elements of `y` to the components of `dy`.

The asynchronous statement in line 19 multiplies `dx` and `dy` and assigns the result to `dz`.

In line 20, the result of the postfix unary operator `[+]` is distributed over `s`. All its components are equal to the sum of all components of the operand `dz`. Here, the result of the prefix unary operator `[host]` is the component of the operand belonging to the host-processor. So, the statement in line 20 assigns the sum of all components of `dz` to `z`. Note, that here we could replace the operator `[host]` by the operator `[s:I==0]` with the same result.

Finally, line 21 performs calling `sqrt` on the host-processor and assigning the value returned to `z`.

To support modular parallel programming as well as the writing of a library of parallel programs, so-called network functions are introduced in addition to basic and nodal functions.

Network function is called and executed on some network or subnetwork, and its value (if any) is also distributed over this region of the computing space. The header of the definition of the network function either specifies an identifier of global static network or subnetwork, or declares an identifier of network being a special formal parameter of the function. In the first case, the function can be called only on the specified region of the computing space. In the second case, it can be called on any network or subnetwork of a suitable type. In any case, only the network specified in the header of the function definition may be used in the function body. No network can be declared in the body, but subnetworks of the network specified in the header can be declared. Only data objects belonging to the network specified in the header may be defined in the body. In addition, corresponding components of an externally-defined distributed data object may be used.

When programming in mpC, networks are used first of all for adequate decomposition of user's problem. In fact, every virtual processor corresponds to a task that may run in parallel. Static and automatic networks allow to specify static and dynamic structures of interacting tasks. Networks also allow the user to specify the tasks in arbitrary manner mixing data and task parallelism in the most appropriate to his problem way. Besides, automatic networks allow to specify tasks living in disjoint intervals of time that allows the compiler to place different tasks on the same actual computing node without loss of efficiency. In addition, automatic networks allow to hide a parallelism into basic functions supporting the development of such a library of parallel programs that can be used in the sequential style. Finally, detailed network-topology declarations are the very machine-independent specifications that allow the compiler to map virtual processors optimally to particular processors.

Note, that mpC supports such a style of parallel programming, when synchronization of parallel computations is responsibility of the compiler, not of the user.

3. Programming environment in brief

Currently, the mpC programming environment includes a compiler, a run-time support system (RTSS), a small library, a detector of DMM's topology, and a non-graphical user interface.

The main function of the compiler is to translate an mpC program into a set of programs, each of which runs on its own (virtual) processor, and which in total implement the computations and communications specified by the initial mpC program interacting by means of message passing. The target language is optionally C[] or C (depending on the compiler mode) with calls to functions of RTSS. The compilation unit is an mpC file.

RTSS manages the computing space which consists of a number of processes running over target DMM (currently, a network of workstations) as well as provides communications. It has a precisely specified interface and encapsulates a particular communicating package (currently, a small subset of MPI). It ensures platform-independence of the rest compiler components.

The library consists of a small number of functions which support debugging mpC programs as well as provide some low-level efficient facilities.

The topology detector executes a special test program to detect performances of workstations constituting the target DMM, the number of processors in each of these workstations, as well as bandwidths of links connecting the workstations (optionally).

The user interface consists of a number of programs supporting the creation of a virtual parallel machine and the execution of mpC programs on the machine. While creating the machine, its topology is detected by the topology detector and saved in the file used by RTSS. Currently, it works on networks of UNIX workstations running MPI.

4. Resource management implementation

Currently, our compiler uses optionally either the SPMD model of target code, where all processes constituting a target message-passing program run the identical code, or a quasi-SPMD model, where it translates the source mpC file into 2 separate target files - the first for the host-processor and the second for the rest of processors.

All processes constituting the target program are divided into 2 groups - the special process named *dispatcher* playing the role of the computing space manager, and general processes named *nodes* playing the role of processor nodes of the computing space. The special node named *host* is separated. The dispatcher works as a server accepting requests from nodes. The dispatcher does not belong to the computing space.

Running of the target program begins from call to the initialization function `MPC_Init` by all processes constituting the program (including the dispatcher). The initialization includes the following actions:

- initializing the message-passing platform on which RTSS is based;
- separating the dispatcher and determining the host;
- local initialization of nodes that for every node includes initializing data necessary to interact with RTSS such as a local variable containing the node global number in the computing space and so on;
- dispatcher initialization.

The dispatcher is the only process that does not leave the `MPC_Init` function. It loops inside this function accepting requests. A particular example of such request is a program termination.

In the target program, every network or subnetwork of the source mpC program is represented by a set of nodes named *region*. So, 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 predefined host-processor. Thus, just after initialization, the computing space is represented by the host and a set of temporarily free (unemployed) nodes. At any moment of time, a free node either takes part in overall computations or calls to the function `MPC_Offer` waiting for a job.

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, the descriptor `rd` allows the node to get full information about the region as well as identify itself in the region. In the current implementation, the region descriptor holds the following data:

- MPI's communicator implementing the region;
- 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;
- an integer array containing coordinates of the given node in the corresponding network or subnetwork;
- an indicator saying whether the region is static or automatic;
- a pointer to the list of subregions of the given region;
- some additional and /or redundant information aimed at optimization of computations and communications.

For every network-type definition in the source mpC program, the compiler generates 6 topological functions that are used in run time to compute diverse topological information about a network object of a relevant type. The first function returns the total number of nodes in the network object. Since RTSS uses a linear numeration of nodes from `0` to `n-1`, where `n` is the total number of nodes, the second and third functions convert coordinates of a node into its linear number and vice versa. This linear numeration is determined by lexicographic ordering on the set of coordinates of processor nodes. The fourth function returns the linear number of the parent node. The fifth function returns the type and the relative performance of specified node. Finally, the sixth returns the length of directed link connecting a pair of specified nodes. Note, that if for some pair of nodes the network-type definition does not specify any link, this function returns some value corresponding to a very long link.

If the region represents a network, creation of the region involves the parent node, the dispatcher and all free nodes. The parent node calls to the function `MPC_Net_create` to send the creation request containing the necessary information about the network topology to the dispatcher. Based on this information, the dispatcher selects the most appropriate set of free nodes. After that, it sends to every free node a message saying whether the node is hired in the created region or not. In addition, for the nodes hired in the region this message contains the information necessary to complete initialization of the region descriptor. Meantime, all free nodes call to the function `MPC_Offer` waiting for the message from the dispatcher. After returning from these functions, all the nodes constituting the region get the corresponding region descriptor initialized completely. Deallocation of network region involves all its members calling to the function MPC_Net_free as well as the dispatcher.

When the dispatcher selecting free nodes for the network region to be created, it firstly calls to the corresponding topological functions to compute relative performances of virtual processors of the corresponding network as well as lengths of links between them. The relative performance of a virtual processor is represented by a positive floating number, the parent virtual processor having the relative performance equal to 1. After, the dispatcher computes the absolute performance of each virtual processor by multiplying its relative performance by the parent virtual processor absolute performance.

On the other hand, the dispatcher holds a map of the computing space reflecting its topological properties. The initial state of the map is formed during the dispatcher initialization and contains the following information:

- the number of actual computing nodes and their performances;

- a set of nodes associated with each of the actual computing nodes;

- for every pair of actual computing nodes, the time of message-passing initialization and the time of passing one byte;

- for every actual computing node, the time of message-passing initialization and the time of passing one byte between a pair of processes both running on the actual computing node.

In general, performance of an actual computing node is characterized by two attributes. The first says how quickly a single process runs on the actual computing node, and the second named *scalability* says how many non-interacting processes may run on the actual computing node in parallel without loss of speed. The latter attribute is useful, for example, if the actual computing node is a multi-processor Sun SPARCstation.

Currently, the selection procedure performed by the dispatcher is based on the following simplest scheme.

The dispatcher assigns a weight, being a positive floating number, to every virtual processor of the allocated network in such a way that:

- the more powerful is the virtual processor, the greater weight it obtains;

- the shorter are the links outgoing from the virtual processor, the greater weight it obtains.

These weights are normalized in such a way, that the host-processor is of the weight equal to 1. Similarly, the dispatcher transforms the map of the computing space in such a way, that every actual computing node obtains a weight, being a positive floating number characterizing its computational and communicational power when running a single process.

The dispatcher appoints free nodes of the computing space to virtual processors successively, starting from the virtual processor having the greatest weight `v`. It tries to appoint the most powerful free node to this virtual processor. To do it, the dispatcher estimates the power of a free node for each actual computing node as follows. Let `w` be a weight of the actual computing node `P`, and `N` be a scalability of `P`. Let the set `H` of hired nodes associated with `P` be divided into `N` subsets `h[1]`,...,`h[N]`. Let `v[i][j]` be a weight of the virtual process currently occupying `j`-th node in `h[i]`. Then the estimation of the power of a free node associated with `P` is equal to

$$\max_i \left\{ \frac{w}{v + \sum_j v[i]\,[j]} \right\}$$

Thus, if the estimation is maximum for the actual computing node `P`, the dispatcher appoints some free node, associated with `P`, to the virtual processor and adds this virtual processor to `h[k]` such that

$$\frac{w}{v + \sum_j v[k]\,[j]} = \max_i \left\{ \frac{w}{v + \sum_j v[i]\,[j]} \right\}$$

The above procedure appears good enough for networks of workstations and ensures an optimal allocation, if the number of virtual processors in the created network object is not greater than the number of virtual processors having free nodes.

If the region represents a subnetwork, its creation involves only members of the enclosing region which calls to the function `MPC_Subnet_create`. Deallocation of subnetwork region involves only its members calling to the function `MPC_Subnet_free`.

The dispatcher keeps a queue of creation requests. Every time when the queue is not empty, but the dispatcher does not find there a request that can be served, it calls to the deadlock-detection function. The function detects such a situation when the sum of the number of free nodes and the number of such hired nodes which could be released is less than the minimum number of free nodes required by a request in the queue. By definition, a node can be released, if and only if:

- it does not belong to a static network region;

- for each automatic network region, to which it belongs, in the queue there is no unserved creation request generated by the region;

- for each automatic network region, whose creation was initiated directly or indirectly by some automatic network region to which the node belongs, in the queue there does not exist an unserved creation request generated by the region.

The situation is really a deadlock, since even if sometime all the nodes, which could be released, will be actually released, no creation request that was in the queue, when the dispatcher called to the deadlock-detection function, can be served, and appearance of new requests in the queue cannot change this situation positively.

To support detecting a deadlock, the dispatcher associates a creation-request counter with every node of the computing space. Every time, when an automatic-network-creation request sent by a node appears in the queue, the dispatcher firstly increments the value of the counter for all nodes of all automatic network regions including the node, afteron does it for all automatic-network regions which have initiated the creation of automatic-network regions including the node, and so on up to the host-node.

After the creation request is served, all these counters are decremented. A node, whose creation-request counter is non-zero, cannot be released. Static-network-creation requests are counted separately. The nodes constituting a static-network region are simply removed from the set of releasable nodes.

When deadlock is detected, the dispatcher calls to the function `MPC_Abort` which terminates the program. To minimize the probability of occurring a deadlock, the dispatcher tries to serve first automatic-network-creation requests from the node furthest from the host-node.

Note, that although currently the dispatcher is implemented as a single process, the RTSS interface allows one to implement it as a distributed component also.

5. Illustrative example

Let us consider an application studying the evolution of a system of stars in a galaxy (or set of galaxies) under the influence of Newtonian gravitational attraction. It is a classical N-body simulation in which every body exerts forces on all others. The simulation proceeds over a large number of time-step computing the net force on every body and updating its position and other attributes. The most time-consuming phase in every time-step is that of computing the interactions among all the bodies in the system. The simplest method to do this computes all pairwise interactions between bodies.

Let our system consist of a few large groups of bodies. 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. Let it be true in our case. So, we can parallelize the problem, and our application will use a few virtual processors, each of which updates data characterizing a single group of bodies. Each virtual processor holds attributes of all the bodies constituting the corresponding group as well as masses and centers of gravity of other groups. The attributes characterizing a body include its position, velocity and mass.

Finally, let our application allow both the number of groups and the number of bodies in each group to be defined in run time.

The application implements the following scheme:

```
Initializing the galaxy
Scattering groups over processors
Parallel computing masses of groups
Interchanging the masses among
   processors
while(1) {
  Visualization of the galaxy
  Parallel computation of centers of
     gravity of groups
  Interchanging the centers among
     processors
  Parallel updating groups
  Gathering groups
}
```

The fragments of the corresponding mpC program, related with managing the computing space, may look as follows:

```
...
/*The maximum number of groups*/
#define MaxGs 30

/*The maximum number of bodies in a group*/
#define MaxBs 600

/*The number of groups*/
int [host]M;

/*The numbers of bodies in groups*/
int [host]N[MaxGs];

repl dM, dN[MaxGs];

/*Bodies of a galaxy*/
Body (*[host]Galaxy[MaxGs])[MaxBs];

nettype GalaxyNet(m, n[m]) {
  coord I=m;
  node {
    I>=0: fast*n[I] scalar;
  }
  link (J=m){
    J>0: length*(-1) [J]->[0];
    J>0: length*1    [I]->[J];
  }
};
...
void [*]Nbody(char *[host]infile)
{
  /*Initializing Galaxy, M and N*/
  Input(infile);

  /*Broadcasting the number of groups*/
  dM=M;

  /*Broadcasting the numbers of bodies*/
  /*in groups*/
  dN[]=N[];
  {
    net GalaxyNet(dM,dN) g;
    ...
  }
}
```

The network `g`, executing computations and communications, is defined in such a way, that the more powerful is the virtual processor, the larger group of bodies it computes, and the more intensive is the data transfer between two virtual processors, the shorter link connects them (the length specifier `length*(-1)` specifies a shorter link than `length*1` does). The mpC programming environment bases on this information to distribute the virtual processors among actual computing nodes. Since it does it in run time, you need not recompile the mpC program, when transferring it to another DMM. Thus, mpC and its programming environment allow to write dynamic-topology applications which once compiled will run efficiently on any DMM.

We compared the running time of our mpC program to its carefully written MPI counterpart both modelling the 15-hour evolution of 9-group galaxy. We used 3 workstations - SPARCstation 5 (hostname `gamma`), SPARCclassic (`omega`), and SPARCstation 20 (`alpha`), connected via 10Mbits Ethernet. There were 23 other computers in the same segment of the local network. We used LAM MPI Version 5.2 [13] as a particular communication platform.

The computing space of the mpC programming environment consisted of 15 processes, 5 processes running on each processor. The dispatcher ran on `gamma` and used the following relative performances of the processors obtained automatically upon the creation of the virtual parallel machine: 1150 (`gamma`), 331 (`omega`), 1662 (`alpha`).

The MPI program was written in such a way to minimize communication overheads. We mapped 3 MPI processes on `gamma`, 1 process to `omega`, and 5 processes to `alpha`, providing the optimal mapping if the numbers of bodies in the groups are equal to each other.

For homogenous input data when all groups consist of the same number (>100) of bodies, the running time of the MPI program was about 95-98% of the running time of the mpC program.

For heterogenous input data when our groups consisted of 10, 10, 10, 100, 100, 100, 600, 600, and 600 bodies correspondingly, the MPI program took from 88 to 391 seconds depending on the order of these numbers. The running time of the mpC program did not depend on the order of the numbers and was equal to 94 seconds. In any case, the dispatcher selected:

- 4 processes on `gamma` for virtual processors of network `g` computing two 10-body groups, one 100-body group, and one 600-body group;
- 3 processes on `omega` for virtual processors computing one 10-body group and two 100-body groups;
- 2 processes on `alpha` for virtual processors computing two 600-body groups.

6. Regular problems

The above example has demonstrated that mpC and its programming environment allows the user to write efficient and portable programs solving irregular problems. At the same time, they also allow efficient and portable programming regular problems.

Let us consider how one can write in mpC the multiplication of two dense square matrices. The corresponding program may implement the following scheme:

- input the dimension `n` of matrices on the host;
- input two source `nxn` matrices `x` and `y` on the host;
- create a network executing the matrix multiplication;
- scatter rows of the matrix `x` from the host to virtual processors of the network in such a way that the more powerful the virtual processor, the greater number of rows it obtains;
- broadcast the matrix `y` to all virtual processors of the network;
- compute the corresponding rows of the resulting matrix in parallel;
- gather the computed rows of the resulting matrix from the virtual processors of the network to the host. The main peculiarity of the program, which ensures its efficiency and portability, concludes in the following. Before the creation of the network, the program calls to the nodal library function `MPC_Processors_static_info` returning the number `N` and the relative performances `p[0],...,p[N-1]` of actual computing nodes which make up a particular DMM. Then, it creates such a network that consists of N virtual processors with the relative performances `p[0],...,p[N-1]` correspondingly. The dispatcher will map each virtual processor of this network on the corresponding actual computing node, ensuring the most efficient execution of the program on the particular DMM.

More details about programming regular problems in mpC can be found in [18].

7. Summary

The key peculiarity of the mpC language is its advanced facilities for managing such resources of DMMs as processors and links between them. The user can manage these resources in the manner similar to managing the storage in the C language. These facilities allow to develop parallel programs for DMMs that once compiled will run efficiently on any particular DMM, because the mpC programming environment ensures optimal distribution of computations and communications over the DMM in run time. It makes mpC and its programming environment suitable tools for development of a library of parallel programs, especially for heterogenous DMMs.

This paper has presented some details of the implementation of these facilities in the framework of the prototype programming environment developed in the Institute for System Programming, Russian Academy of Sciences.

8. Acknowledgments

The work was supported by ONR and partially by Russian Basic Research Foundation.

9. References

[1] V. Sunderam, "PVM: A framework for parallel distributed computing", *Concurrency: Practice and Experience,* 2(4), 1990, pp.315-339.

[2] Message Passing Interface Forum, "MPI: A Message-passing Interface Standard", *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[3] I. Foster, and K. M. Chandy, *Fortran M: a language for modular parallel programming. Preprint MCS-P327-0992*, Argonne National Lab, 1992.

[4] K. M. Chandy, and C. Kesselman, *CC++: A Declarative Concurrent Object Oriented Programming Language. Technical Report CS-TR-92-01*, California Institute of Technology, Pasadena, California, 1992.

[5] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu, *Fortran D Language Specification. Center for Research on Parallel Computation*, Rice University, Houston, TX, October 1993.

[6] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran", *Scientific Programming*, 1(1), 1992, pp.31-50.

[7] High Performance Fortran Forum., *High Performance Fortran language specification, version 1.0*. Rice University, Houston, TX, May 1993.

[8] *CM-5 Technical Summary. The CM Fortran Programming Language*, Thinking Machines Corp., November. 1992.

[9] *CM-5 Technical Summary. The C* Programming Language*, Thinking Machines Corporation, November 1992.

[10] P. J. Hatcher, and M. J. Quinn, *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

[11] M. Philippsen, and W. Tichy, "Modula-2* and its compilation", *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, 1991.

[12] P. J. Hatcher, and M. J. Quinn, "Supporting Data-Level and Processor-Level Parallelism in Data-Parallel Programming Languages", *Proceedings of the 26th Annual Hawaii International Conference on System Sciences, Volume II, Software Technology*, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 14-22.

[13] *Trollius LAM Implementation of MPI. Version 5.2*. Ohio State University, 1994.

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

[15] A. Lastovetsky, *The mpC Programming Language Specification. Technical Report*, Institute for System Programming, Russian Academy of Sciences, Moscow, 1994.

[16] D. Arapov, A. Kalinov, and A. Lastovetsky, "Managing the Computing Space in the mpC Compiler", *Proceedings of the 1996 Parallel Architectures and Compilation Techniques (PACT'96) conference*, IEEE Computer Society Press, Boston, October 1996.

[17] S. Gaissaryan, and A. Lastovetsky, "ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler", *Journal of C Language Translation*, 5(3), 1994, pp.183-198.

[18] D. Arapov, V. Ivannikov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis, "A Parallel Language for Modular Distributed Programming", *Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs'97)*, IEEE Computer Society Press, Aizu-Wakamatsu, Japan, March 1997 (to appear).